ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA
SCUOLA DI SCIENZE

CORSO DI LAUREA IN INGEGNERIA E SCIENZE INFORMATICHE

CURVE SPLINE PER L'ANIMAZIONE DIGITALE

Relazione finale in
Computer Graphics

Relatrice
Prof.ssa Serena Morigi

Presentata da
Marco Galassi

Correlatrice
Dott.ssa Damiana Lazzaro

Sessione: II
Anno Accademico: 2016/2017

Indice

Introduzione

Lo scopo di questa tesi è stato lo studio di tecniche numerico-grafiche e la realizzazione di un software per l'animazione digitale di navigazione lungo un percorso. Le tematiche coperte spaziano dagli algoritmi numerici e di modellazione numerica alle tecniche algoritmiche di computer graphics. La motivazione per cui si è deciso di creare un software standalone anziché appoggiarsi a svariati software presenti sul commercio merita una considerazione. I passi da gigante fatti dallo sviluppo della computer grafica negli ultimi due decenni hanno permesso a grandi aziende, piccoli sviluppatori o semplici appassionati di potersi cimentare nella realizzazione di animazioni, modellazioni, film e videogiochi con estrema facilità. Se in passato tutto questo poteva essere realizzato con un costo in tempo e in denaro decisamente proibitivo per molti, ora la situazione è drasticamente cambiata. Sono disponibili tanti tipi diversi di software che possono essere utilizzati con licenza gratuita e si possono reperire un numero enorme di tutorial che ci possono guidare, passo dopo passo, al completamento dei nostri progetti. La conseguenza principale della diffusione di nuovi motori grafici gratuiti è senza dubbio la nascita di una fiorente comunità online dove la maggior parte dei nostri dubbi in fase di sviluppo può trovare risposta. Armato di sufficiente pazienza, chiunque può togliersi delle soddisfazioni.
A questi notevoli fatti positivi se ne contrappone uno che ritengo essere spiacevolmente controproducente e che ha contribuito e motivato il tipo di progetto per la mia tesi. La necessità di doversi mostrare (per ragioni spesso di mercato o di pubblicità) *easy-to-use* e rapidamente accessibile a qualunque tipo di utente ha spinto i designer dei più popolari motori grafici ad adottare una filosofia ben precisa: nascondere all'utente la natura matematica delle componenti che decide di utilizzare. Questa scelta è assolutamente comprensibile e io stesso confesso di essermi trovato in più di una occasione scoraggiato dalla difficoltà di alcuni argomenti. Tuttavia, da grande appassionato di opere videoludiche, ho sempre avuto una certa curiosità riguardo le tecniche usate in questo campo. Inutile spiegare la mia soddisfazione quando ho potuto approfondire questi stessi concetti durante le lezioni di Algoritmi Numerici e Computer Graphics. Ho pertanto deciso di spingermi un po' più a fondo e provare a realizzare - con il solo appoggio di librerie open source - un motore grafico basilare. Il mio obiettivo è stato pertanto quello di realizzare un progetto che mi permettesse di

venire a conoscenza di particolari algoritmi e strutture dati usate dagli esperti di computer grafica. Quello che solitamente viene dato per scontato da chi si serve di questi tipi di *engine* nasconde spesso dietro di sé un'interessante complessità. Comprendere e fare miei questi argomenti è stato qualcosa di incredibilmente soddisfacente.

Dal punto di vista concreto, il progetto si compone essenzialmente di due parti.

La prima parte verte sullo studio di determinate curve parametriche dette B-spline. Ampiamente studiate da oltre 30 anni, esse vengono largamente utilizzate nella modellazione di superfici e traiettorie che si richiede siano dotate di alcune caratteristiche, prima fra tutte la forma liscia e sinuosa. Il software permette all'utente di definire a piacere una curva che verrà percorsa dalla telecamera durante l'animazione.

La seconda parte è invece dedicata all'implementazione delle più conosciute tecniche di animazione lungo un percorso. Un motore grafico, anche se basilare, deve necessariamente soddisfare alcune richieste dell'utente, quali, in primo luogo, permettere di scegliere l'obiettivo della telecamera e modificare a piacere la velocità di percorrenza della traiettoria da parte di quest'ultima.

Come detto precedentemente, il software fa uso di diverse componenti open source. La libreria grafica di base utilizzata è OpenGL (integrata con FreeGLUT), mentre i modelli della scena sono stati realizzati su Blender e poi successivamente importati. Sì è inoltre rivelata particolarmente utile la libreria matematica GSL (GNU Scientific Library).

Questo documento è strutturato per capitoli. Il capitolo 1 si dedica alla spiegazione della teoria matematica necessaria per comprendere il funzionamento del software. Parlerò nello specifico di curve spline e delle loro proprietà e si analizzeranno i problemi di interpolazione e approssimazione di punti. Il capitolo 2 descrive le tre componenti di una generale animazione di *camera motion*: percorso, velocità e orientamento della telecamera. Discuterò inoltre delle varie tecniche utilizzate nel progetto CaMot. Il capitolo 3 si concentra sulla parte di modellazione degli oggetti che compongono la scena e comprende una breve panoramica sul controllo delle collisioni. Il software viene descritto nelle sue parti più importanti nel corso del capitolo 4. Questo comprende un'esaustiva spiegazione dell'interfaccia utente e di tutto ciò che è stato utilizzato durante lo sviluppo.

Capitolo 1

Curve spline

1.1 Spline e B-Spline

Una spline è una funzione matematica costituita da un insieme di polinomi dello stesso grado raccordati fra loro che ha lo scopo di interpolare oppure approssimare efficacemente un insieme di punti, al fine di avere una forma liscia e di non possedere bruschi cambiamenti alla propria pendenza. Questo condizioni si rivelano particolarmente importanti in molti ambiti della computer grafica e il software ne mostra forse il più diffuso: la creazione della traiettoria che la telecamera deve seguire nel corso di un'animazione. La forma che decidiamo influisce pesantemente sulla qualità finale e se non curiamo questo aspetto potremmo trovarci con degli effetti indesiderati.

1.1.1 Breve introduzione sulle curve parametriche

Il software da me realizzato fa un uso intensivo di curve parametriche e questo lo si può immediatamente vedere nella finestra che controlla la velocità della telecamera e dalla traiettoria che quest'ultima segue durante l'animazione. Ritengo saggio fare una piccola premessa sulle cose più importanti da tenere in considerazione quando si parla di questo tipo di curva, in quanto userò varie volte questo termine nel corso della tesi. Le spline sono curve parametriche e pertanto a loro si applicano tutti i concetti spiegati di seguito.
Iniziamo con il dire che una curva parametrica generalmente non è una funzione. Una curva parametrica in R^n è un applicazione

$$C(t) : I = [a, b] \subset R \to R^n$$

$$t \to (\phi_1(t), \phi_2(t), ..., \phi_n(t))$$

dove $\phi_1(t), ..., \phi_n(t)$ sono funzioni continue del parametro $t \in I \subset R$ che vengono dette *componenti parametriche* della curva. Esse sono funzioni reali con variabile reale, o più semplicemente funzioni scalari. Un modo alternativo - che userò più frequentemente - di definire la curva è il seguente:

$$C(t) = \begin{pmatrix} \phi_1(t) \\ \phi_2(t) \\ \vdots \\ \phi_n(t) \end{pmatrix}$$

L'immagine di I tramite C, ovvero $C(I)$, prende il nome di *supporto* o *traiettoria* della curva. $C(t)$ è invece la *parametrizzazione*. Per quanto possano sembrare simili, supporto e parametrizzazione non sono la stessa cosa. Per la precisione, curve con lo stesso supporto possono avere parametrizzazioni diverse. Il seguente esempio sarà utile per chiarire i dubbi.
Consideriamo una particella che si muove nel tempo lungo una circonferenza. Se indichiamo t come istante di tempo, possiamo scrivere la stessa curva in due modi diversi:

$$C(t) = (\cos(t), \sin(t)) \qquad t \in [0, 2\pi]$$

$$C(t) = (\cos(2t), \sin(2t)) \qquad t \in [0, 2\pi]$$

La differenza è solo nella velocità con cui la circonferenza viene percorsa: la seconda permette alla particella di percorrere due volte l'intera curva.
L'esempio risulta calzante anche per discutere dell'*equivalenza* tra due curve. In maniera formale, due curve $C_1(t) : I \to R^n$ e $C_2(t) : J \to R^n$ si dicono equivalenti se esiste una funzione $g : I \to J$ invertibile e con derivate mai nulle, tali che:

$$C_1(t) = C_2(g(t)) \qquad t \in I$$

Si dice in questo caso che C_1 si ottiene da C_2 per *riparametrizzazione*. Le curve nell'esempio sono equivalenti: è infatti possibile prendere la funzione invertibile e con derivata mai nulla $g = 2t$ per passare dalla prima curva alla seconda. Altre parametrizzazioni interessanti verranno discusse a parte.
Le curve spline sono curve parametriche $s(t) : I = [0, 1] \subset R \to R^2$ oppure $s(t) : I = [0, 1] \subset R \to R^3$ con componenti:

$$s(t) = \begin{pmatrix} x(t) \\ y(t) \end{pmatrix} \qquad e \qquad s(t) = \begin{pmatrix} x(t) \\ y(t) \\ z(t) \end{pmatrix}$$

Lunghezza di una curva parametrica

Consideriamo una curva $C(t) : [a, b] \to R^n$ di cui vogliamo conoscere la lunghezza totale L. Immaginiamo di avere una sequenza di punti di valutazione di ordine crescente $P = \{a = t_1 < ...t_n = b\}$ che formano una spezzata. La lunghezza di tale sequenza è pari a

$$L(P) = \|C_2 - C1\|_2 + ... + \|C_n - Cn - 1\|_2 = \sum_{i=1}^{n-1} \|C_{i+1} - Ci\|_2$$

Consideriamo ora P^1, ovvero P alla quale aggiungiamo un punto. La spezzata si avvicina sempre più alla curva e questo garantisce che $L(P^1) \geq L(P)$. Iterando questo procedimento si arriverà ad un'approssimazione sempre più fedele della curva:

$$L = lim_{i \to \infty} L(P^i)$$

Geometricamente parlando, stiamo sommando delle valutazioni in diversi punti della derivata della curva. Trasformando il limite di una sommatoria in un integrale, possiamo concludere che la lunghezza di una qualsiasi curva parametrica $C(t)$ è pari a:

$$L = \int_a^b \|C't)\|_2 dt = \int_a^b \sqrt{C'(t)} dt$$

In generale, prese due coppie qualsiasi di punti sulla curva tra loro consecutivi P_i, P_{i+1} e P_j, P_{j+1} con $i \neq j$, la distanza definita in quei due intervalli non sarà la stessa. Risulta pertanto che

$$\int_i^{i+1} \|C'(t)\|_2 dt \neq \int_j^{j+1} \|C'(t)\|_2 dt \qquad i \neq j$$

1.1.2 Definizione di funzione spline

Dato un intervallo chiuso $[a, b]$, consideriamo una partizione dello spazio Δ dato da diversi punti x_i tali per cui $\Delta = \{a = x_0 < x_1 < x_2 < ... < x_k < x_{k+1} = b\}$ che chiameremo *partizione nodale*. Questi punti prendono il nome di *nodi* e sono in totale $k+2$. Chiamiamo per comodità *veri* i k nodi che non coincidono con gli estremi. Definiamo inoltre con m l'ordine dei polinomi che compongono la curva ($m < k$).
Una spline di ordine m, indicata con $s_m(x)$, che ha come nodi i punti x_i è una funzione definita su $[a, b]$ tale che soddisfi le seguenti condizioni:

1. In ogni intervallo $[x_i, x_{i+1}]$ con $i = 0, 1, ..., k$ la funzione $s_m(x)$ è un polinomio di ordine m.
2. La funzione $s_m(x)$ e le sue prime $m - 2$ derivate sono continue sui nodi.

La seconda condizione ci assicura che i nodi sono i punti di contatto fra due polinomi e allo stesso tempo che essi si congiungano in maniera flessibile (*smooth*). Essa può essere scritta formalmente nel seguente modo:

$$\frac{d^j p_{i-1}(x_i)}{dx^j} = \frac{d^j p_i(x_i)}{dx^j} \qquad \text{per} \quad i = 1, ..., k \quad \text{e} \quad j = 0, .., m-2$$

dove p_i è il polinomio definito nell'intorno $[x_i, x_{i+1}]$. Il numero di polinomi che compongono la spline sono in totale $k+1$.

1.1.3 Definizione dello spazio spline a nodi semplici

Consideriamo ora lo spazio $S_m(\Delta)$ e ci chiediamo quale sia la sua dimensione. Sappiamo che i nodi individuano esattamente $k+1$ polinomi di ordine m. Questi polinomi, indipendentemente dalla base scelta, necessitano di un numero di coefficienti pari al loro ordine per poter essere rappresentati. Questo ci porta ad una dimensione dello spazio delle spline pari a $(k+1)m$. Esistono tuttavia le condizioni di continuità da rispettare, che tolgono dei gradi di libertà alla scelta dei coefficienti e quindi abbassano la suddetta dimensione. Le condizioni di continuità, da applicare a qualsiasi tipo di spline, sono in totale pari a $(m-1)k$. Questo significa che generalmente una spline ha un grado di libertà pari a $(k+1)m - (m-1)k = km + m - mk + k = m + k$.
Chiamiamo generalmente la base di questo spazio *B-spline*. Preciso subito che in letteratura c'è spesso ambiguità nell'utilizzo di questo termine, poiché viene utilizzato anche per definire le curve stesse generate in modo stabile da questa base. Diamo ora una definizione formale.
Data una partizione nodale Δ, definiamo con $N_{i,m}(x)$ la funzione B-spline normalizzata di ordine m in questo modo:

$$N_{i,m}(x) = \frac{x - x_i}{x_{i+m-1} - x_i} N_{i,m-1}(x) + \frac{x_{i+m} - x}{x_{i+m} - x_{i+1}} N_{i+1,m-1}(x) \qquad \text{con } x_i \leq x \leq x_{i+m}$$

Notiamo subito che si tratta di una funzione ricorsiva data dalla somma di due funzioni di ordine più basso. Alla base di ricorsione (ordine 1) troviamo:

$$N_{i,1}(x) = \begin{cases} 1, & \text{se } x_i \leq x \leq x_{i+1} \\ 0, & \text{altrimenti} \end{cases}$$

Si deve la proposta di queste formule a de Boor[1] e Cox.

[1] Carl R. de Boor. *A Practical Guide to Splines, Revised Edition*. 2003.

Estensione del vettore dei nodi

Poiché le funzioni B-spline rappresentano una base per lo spazio $S_m(\Delta)$ una qualsiasi combinazione lineare di queste genera una spline. Se $s(x) \in S_m(\Delta)$ allora:

$$s(x) = \sum_{j=1}^{m+k} c_j N_{j,m}(x)$$

Proviamo ad immagine graficamente la forma di queste B-spline. Questa dipende essenzialmente dall'ordine m che la definisce. In generale, abbiamo bisogno di esattamente $k+m$ funzioni per determinare la curva in ogni suo punto. Ogni funzione di ordine m è definita in esattamente m intervalli continui. Questo ci pone in difficoltà: avendo solo k nodi veri, possiamo creare solo $k-m$ b-spline.
La soluzione a questo problema consiste nella definizione di un nuovo vettore di nodi, che definiamo *esteso*. Questo possederà un numero di nodi fittizi che, aggiunti al numero di quelli veri già esistenti, permetterà di disegnare ogni funzione. I nodi aggiuntivi vengono solitamente posti a sinistra e a destra dei nodi veri, in egual numero. Porli coincidenti con gli estremi oppure oltre di essi (mantenendo sempre l'andamento crescente) varierà la forma della curva.
Come cambia il nostro vettore $\Delta = \{a = x_0 < x_1 < ... < x_k < x_{k+1} = b\}$? Verranno aggiunti esattamente $m-1$ nodi fittizi a sinistra di a e lo stesso numero a destra di b. Questi nodi, come detto precedentemente, possono anche coincidere rispettivamente con gli estremi $a = x_0$ e $b = x_{n+1}$. Definiamo ora espressamente il vettore esteso che chiameremo *partizione nodale estesa* e vediamo la relazione che esiste con Δ. Lo indichiamo con $\Delta^* = \{t_{-m+1} \leq ... \leq b = t_0 < t_1 < ... < t_k < t_{k+1} = a \leq ... \leq t_{k+m}\}$. Presento inoltre anche una notazione alternativa più compatta: $\Delta^* = \{t_i\}_{i=-m+1}^{k+m}$.
La costruzione di questo vettore è invece la seguente:

$$\Delta^* = \begin{cases} t_i \leq a, & \text{se } -m+1 \leq i \leq 0 \\ t_i \equiv x_i, & \text{se } 1 \leq i \leq k \\ t_i \geq b, & \text{se } k+1 \leq i \leq k+m \end{cases}$$

Una qualsiasi spline $s(x) \in S_m(\Delta^*)$ può quindi essere definita come una nuova combinazione lineare che tiene in considerazione i nuovi nodi:

$$s(x) = \sum_{j=-m+1}^{k} c_j N_{j,m}(x)$$

Le $m+k$ B-spline generatrici vengono definite a partire dal nodo t_{-m-1}.

Proprietà delle B-spline

Le B-spline godono di alcune proprietà particolari:

1. Per ogni punto $\bar{x} \in [a, b]$ è sufficiente valutare esattamente $m - 1$ B-spline se $\bar{x} \equiv x_i$ (ovvero se coincide con un nodo), m altrimenti.
2. $N_{i,m}(x) = 0 \qquad \text{per } x < x_i \quad \text{e} \quad x > x_{i+m}.$
3. $N_{i,m}(x) \geq 0 \qquad \text{per } x_i < x < x_{i+m}.$
4. $\sum_{j=-m+1}^{k} N_{j,m}(x) = 1.$

Le prime due condizioni garantiscono che le B-spline e le curve da esse generate abbiano *supporto compatto* (o *locale*). Questo dà luogo a due importanti conseguenze. La prima consiste nel fatto che la valutazione di un punto $\bar{x} \in [t_l, t_{l+1}]$ non è assolutamente influenzata dalla posizione dei nodi a sinistra di t_{l-m+1} e a destra di t_{l+1}. La seconda è che non è necessario valutare $k + m$ B-spline per conoscere il valore di $s(x)$: ne sono sufficienti solo m. È proprio qui che risiede la straordinaria utilità di queste funzioni base. Il nostro vettore esteso Δ^* potrà contenere un numero altissimo di nodi ma questo non aumenta la complessità computazionale. Possiamo dunque scrivere $s(x) \in S_m(\Delta^*)$ come:

$$s(x) = \sum_{j=l-m+1}^{l} c_j N_{j,m}(x) \quad \text{con} \quad x \in [t_l, t_{l+1}]$$

La terza proprietà è detta *non negatività* mentre la quarta ci dice che lo spazio delle B-spline è a tutti gli effetti un *partizione dell'unità*. L'unione di queste due caratteristiche ci permette di affermare che qualunque spline generata è una combinazione lineare *convessa* dei coefficienti c_j. Come conseguenza sappiamo che:

$$\min c_j \leq s(x) \leq \max c_j$$

Di seguito cito altre due proprietà interessanti. La prima è detta *variation diminishing* e afferma che il numero di volte in cui la spline generata varia di segno è sempre minore od uguale al numero di volte in cui la successione dei suoi coefficienti ¡ varia di segno. La seconda riguarda la definizione dell'intervallo $[a, b]$. Una spline generata è sempre opportunamente normalizzabile e quindi si è solito porre come estremo sinistro 0 e come estremo destro 1. Da questo momento in poi, ad eccezione di dove è necessario il contrario, la successione dei nodi veri $x_i \in \Delta^*$ si ritiene definita nell'intervallo $[0, 1]$.

1.1.4 Definizione dello spazio spline a nodi multipli

Le spline finora descritte sono polinomi di ordine m che hanno continuità fino alle prime $m-2$ derivate. Formalmente esse sono quindi di classe C^{m-2}. Può tuttavia capitare che questa caratteristica risulti svantaggiosa se i dati forniti hanno un andamento che è in parte molto regolare e in parte meno. Per rendere la curva ulteriormente più flessibile possiamo operare direttamente sui punti di contatto dei polinomi e ridurre le condizioni di raccordo sulle derivate. A livello di B-spline questo si riduce ad aumentare la molteplicità sui relativi nodi veri. Una spline dotata di nodi con molteplicità diverse viene detta *a nodi multipli*.
Definiamo ora con più chiarezza questo concetto. Sia $[a,b]$ un intervallo chiuso e limitato e Δ una partizione su di esso di $k+2$ nodi tali che $\Delta = \{a = x_0 < x_1 < ... < x_k < x_{k+1} = b\}$. Come già visto, otteniamo in tutto $k+1$ sottointervalli. Definito l'ordine dei polinomi m, sia $M = (m_1, m_2, ..., m_k)$ un vettore di interi positivi tali che $1 \leq m_i \leq m$, $\forall i = 1, ..., k$. M viene definito *vettore delle molteplicità*.
Si definisce funzione spline polinomiale di ordine m con nodi $x_1, ..., x_k$ e molteplicità $m_1, ..., m_k$ una funzione $s_m(x)$ definita su $[a,b]$ tale che:

1. In ogni intervallo $[x_i, x_{i+1}]$ con $i = 0, 1, ..., k$ la funzione $s_m(x)$ è un polinomio di ordine m.
2. La funzione $s_m(x)$ e le sue prime $m - m_i - 2$ derivate sono continue sui nodi m_i, $\forall i = 1, ..., k$.

La seconda condizione definisce le diverse continuità sui nodi e può essere scritta formalmente nel seguente modo:

$$\frac{d^j p_{i-1}(x_i)}{dx^j} = \frac{d^j p_i(x_i)}{dx^j} \qquad \text{per} \quad i = 1, ..., k \quad \text{e} \quad j = 0, .., m - m_i - 1$$

dove p_i è il polinomio definito nell'intorno $[x_i, x_{i+1}]$. Il vettore M ci permette di decidere la continuità su ogni singolo nodo, ovvero fino a quale derivata effettuare il raccordo. Dalla definizione segue che maggiore è la molteplicità su un nodo e minori saranno le condizioni di continuità. Notiamo infine che se $m_i = 1$, $\forall i = 1, ..., k$ otteniamo una spline a nodi semplici.
Analogamente a quanto fatto nelle sottosezioni precedenti consideriamo la dimensione dello spazio delle spline a nodi multipli $S_m(\Delta, M)$. Ci aspettiamo senza dubbio che tale numero sia maggiore (o al limite uguale) a $m+k$, ovvero la dimensione dello spazio della spline a nodi semplici $S_m(\Delta)$. Se non ci fossero condizioni di continuità potremmo considerare semplicemente il fatto che dobbiamo definire $k+1$ polinomi di grado m e concludere che la dimensione sia $m(k+1)$. Tali condizioni variano da nodo a nodo e sono in totale pari a $\sum_{i=1}^{k}(m - m_i)$. La dimensione dello spazio è pertanto pari a $m(k+1) - \sum_{i=1}^{k}(m - m_i) = mk + m - \sum_{i=1}^{k} m + \sum_{i=1}^{k} m_i = m + \sum_{i=1}^{k} m_i$. Chiamiamo da que-

sto momento in poi $K = \sum_{i=1}^{k} m_i$ e abbiamo trovato intuitivamente che la dimensione dello spazio $S_m(\Delta, M)$ è pari a $m + K$.

Le spline a nodi multipli sono analoghe a quelle semplici e hanno in generale le stesse proprietà. Ogni spline a nodi multipli $s(x) \in S_m(\Delta, M)$ può infatti essere espressa nel seguente modo:

$$s(x) = \sum_{j=1}^{m+K} c_j N_{j,m}(x)$$

dove $N_{1,m}, ..., N_{m+K,m}$ rappresentano la base dello spazio. Abbiamo sempre bisogno di un vettore di nodi esteso per poter determinare suddetta base. Definiamo pertanto una nuova partizione nodale $\Delta^* = \{t_i\}_{i=1}^{2m+K}$ dove tutti i nodi t_i, definiti in una successione crescente, possono anche coincidere. La differenza rispetto a quella vista nella sottosezione precedente riguarda la notazione: l'indice i è tale da essere $1 \leq i \leq 2m + K$ e non più $-m + 1 \leq i \leq m + K$. Questa modifica è lieve e si auspica possa aiutare il lettore nella comprensione. Consideriamo ora i nodi veri $t_{m+1}, ..., t_{m+K}$. Se $K > k$ allora alcuni di questi nodi saranno coincidenti. Il numeri di nodi coincidenti viene dato proprio dalla molteplicità. Come vedremo più avanti, una molteplicità maggiore forza la curva ad avvicinarsi verso il relativo punto di controllo. Questo è la conseguenza diretta della modifica nella definizione delle funzioni base $N_{i,m}, ..., N_{m+K,m}$, che ricordiamo essere pari a:

$$N_{i,h}(x) = \begin{cases} \dfrac{x - t_i}{t_{i+h-1} - t_i} N_{i,h-1}(x) + \dfrac{t_{i+h} - x}{t_{i+h} - t_{i+1}} N_{i+1,h-1}(x) & \text{se } t_i \neq t_{i+h} \\ 0 & \textit{altrimenti} \end{cases}$$

per $h = 2, ..., m$ e

$$N_{i,1}(x) = \begin{cases} 1, & \text{se } t_i \leq x < t_{i+1} \\ 0, & \text{altrimenti} \end{cases}$$

per $h = 1$ (condizione iniziale).

Le altre proprietà di cui godono sia le B-spline che le curve da loro generate sono le stesse discusse precedentemente, con la sola differenza di dover sostituire in ogni occorrenza $k + m$ con $K + n$.

1.1.5 Curve approssimanti di forma

La seguente spiegazione prenderà inizialmente in considerazione spline a nodi semplici, per poi generalizzarsi a quelle a nodi multipli.

Definito l'ordine m, indichiamo con N il numero dei *punti di controllo* $P_i = (x_i, y_i)$, $i = 1, ..., N$ che vogliamo approssimare. La forma della curva è direttamente influenzata dalla posizione di tali punti. Il loro numero è uguale alla dimensione dello spazio delle spline $S_m(\Delta^*)$. Ne consegue

direttamente che:

$$N = k + m$$

dove k è il numero di nodi veri nella partizione estesa $\Delta^* = \{t_i\}_{i=1}^{2m+k}$, ovvero quelli tra t_{m+1} e t_{k+m}. Congiungendo tramite un segmento in successione i punti (P_i con P_{i+1}, $i = 0, ..., N-2$ e P_{N-1} con P_0) si viene a delineare il cosiddetto *poligono di controllo*. La curva si troverà sempre all'interno del poligono e sarà definita dalla combinazione dei punti di controllo. Possiamo pertanto definire:

$$C(t) = \sum_{i=1}^{N} P_i N_{i,m}(t) = \begin{pmatrix} \sum_{i=1}^{N} x_i N_{i,m}(t) \\ \sum_{i=1}^{N} y_i N_{i,m}(t) \end{pmatrix} \qquad t \in [0,1]$$

Ricordo al lettore che le curve godono di tutte le proprietà descritte nelle sezioni precedenti. Sappiamo già pertanto che per calcolare $N_{i,m}(t)$ è sufficiente valutare la funzione in soli m intervalli, precisamente tra i nodi t_{l-m+1} e t_{l+1} se $t \in [t_l, t_{l+1}]$. Questo significa anche che la posizione degli altri nodi è del tutto ininfluente: le modifiche che facciamo ai punti di controllo, grazie al supporto compatto, modificano la curva solo per quei valori determinati sul supporto $I = [t_l, t_{l+1}]$. Ma soprattutto sapere che $0 \leq N_{i,m}(t) \leq 1$, $\forall i$ e che $\sum_{i=0}^{k-1} Ni, m(t) = 1$, $\forall i$ ci dice immediatamente che $C(t)$ è sempre una combinazione lineare convessa dei punti di controllo. Di conseguenza $C(t)$ è sempre compresa nel poligono di controllo e approssima la spezzata.
Esistono due importantissimi algoritmi che desidero spiegare: *Knot Insertion* e l'algoritmo di De Boor. Essi facilitano enormemente la valutazione della curva spline in ogni singolo punto. È lecito supporre che senza la loro diffusione le B-spline avrebbero avuto molto meno successo.

1.1.6 Knot Insertion

L'algoritmo di *Knot Insertion* permette di aggiungere al partizione estesa di nodi $\Delta^* = \{t_i\}_{i=1}^{2m+K}$ un nuovo nodo e allo stesso tempo assicura che la forma della curva e il suo grado rimangano invariata. Il nodo può anche essere inserito in maniera tale da essere coincidente con uno già esistente e questo sarà particolarmente utile. All'incremento di un'unità nella dimensione dello spazio $S_m(\Delta^*, M)$ deve contrapporsi l'aggiunta di un nuovo punto di controllo.
Data una partizione nodale estesa $\Delta^* = \{t_i\}_{i=1}^{2m+K}$ vogliamo sapere che relazione esista tra se stessa e la nuova partizione che si otterrebbe aggiungendo un nodo. Definiamo pertanto un nuovo nodo $\hat{t} \in [t_l, t_{l+1}]$ e lo aggiungiamo al nostro vettore nodale ottenendo $\hat{\Delta}^* = \{t_i\}_{i=1}^{2m+K+1}$. I nodi saranno

così definiti:

$$\hat{t}_i = \begin{cases} t_i & i \leq l \\ \hat{t} & i = l+1 \\ t_{i-1} & i \geq l+2 \end{cases}$$

Il nuovo spazio sarà $S_m(\hat{\Delta}^*, \hat{M})$ e questa porterà alla definizione di nuove funzioni base $\hat{N}_{i,m}$, $i = 1, ..., m+K+1$. La relazione fra queste e quelle dello spazio originale è la seguente:

$$N_{i,m}(x) = \begin{cases} \hat{N}_{i,m}(x) & i \leq l-m \\ \dfrac{\hat{t}-\hat{t}_i}{\hat{t}_{i+m}-\hat{t}_i}\hat{N}_{i,m}(x) + \dfrac{\hat{t}_{i+m+1}-\hat{t}}{\hat{t}_{i+m+1}-\hat{t}_{i+1}}\hat{N}_{i+1,m}(x) & l-m+1 \leq i \leq l \\ \hat{N}_{i+1,m}(x) & i \geq l+1 \end{cases}$$

Possiamo subito notare che non tutte le funzioni base sono cambiate. Solo m hanno subito delle modifiche, ovvero quelle che avevano l'intervallo $[t_l, t_{l+1}]$ come parte del proprio supporto. La nostra attenzione va ora ai coefficienti. Vogliamo individuarne dai nuovi, che denotiamo con $\hat{c}_i$ tali che la curva rimanga uguale, ovvero:

$$s(x) = \sum_{i=1}^{m+K} c_i N_{i,m}(x) = \sum_{i=1}^{m+K+1} \hat{c}_i \hat{N}_{i,m}(x)$$

Questo é sempre verificato se

$$\hat{c}_i = \begin{cases} c_i & i \leq l-m+1 \\ \lambda_i c_i + (1-\lambda_i)c_{i-1} & l-m+2 \leq i \leq l \\ c_{i-1} & i \geq l+1 \end{cases} \qquad \text{con} \quad \lambda_i = \frac{\hat{t}-\hat{t}_i}{\hat{t}_{i+m}-\hat{t}_i} \quad \text{e} \quad (1-\lambda_i) = \frac{\hat{t}_{i+m}-\hat{t}}{\hat{t}_{i+m}-\hat{t}_i}$$

I λ_i sono sempre compresi tra 0 e 1. Questo significa che i nuovi coefficienti sono combinazioni lineare convessa di quelli originali e l'algoritmo si mantiene stabile.
Per ottenere una forma direttamente applicabile alle curve B-spline della suddetta relazione è sufficiente sostituire c_i e $\hat{c}_i$ con P_i e $\hat{P}_i$ rispettivamente.
L'algoritmo di *Knot Insertion* ha due usi principali:

1. È alla base dell'algoritmo di valutazione rapida dei punti sulla curva
2. Permette di spezzare una curva in due separate

L'algoritmo di De Boor verrà spiegato nel dettaglio nella sottosezione seguente. La possibilità di spezzare una curva è una diretta conseguenza dei due algoritmi: se inseriamo tanti nodi coincidenti ad un nodo già esistente $\bar{t}$ in maniera tale da rendere la sua molteplicità pari ad m allora la spline

si divide in due spline definite ognuna sul proprio intervallo, ovvero:

$$s_m(x) = \begin{cases} s1(x) & x \in [a, \bar{t}] \\ s2(x) & x \in [\bar{t}, b] \end{cases}$$

1.1.7 Algoritmo di De Boor

Per ogni singolo punto $\bar{x} \in [0, 1]$ sul quale valutiamo una spline $s_m(\bar{x})$ dovremmo generalmente definire le B-spline che formano lo spazio $S_m(\Delta^*, M)$. Le operazioni che le generano sono gravose dal punto di vista computazionale, anche sapendo che è sufficiente calcolarne solo m grazie al supporto compatto. L'algoritmo di De Boor[2] ci fornisce un metodo stabile per riuscire a valutare un punto di una spline senza dovere valutare le funzioni base.

Abbiamo già discusso di come aumentare la molteplicità di un nodo t_i. Geometricamente questo significa aumentare il grado di libertà su quel punto ovvero diminuire di uno il numero di condizioni di continuità sulle derivate. Un nodo singolo con molteplicità $m_i = 1$ di una spline a nodi semplici di ordine m deve rispettarne in totale $m - 1$. Supponendo di dover utilizzare una spline cubica ($m = 4$), ad ogni nodo risulta una continuità di classe C^2. In generale, la continuità di un nodo t_i di molteplicità $m_i < m$ è di classe C^{m-m_i-1}. Aumentare la molteplicità significa pertanto diminuirne la classe di continuità.

Consideriamo il caso limite $m_i = m-1$, ovvero il massimo di molteplicità che un nodo può raggiungere prima di causare uno spezzamento della curva. Un nodo con questa caratteristica è di classe $C^{m-m_i-1} = C^{m-(m-1)-1} = C^0$. Questo significa che il nodo rispetta solo le condizioni di contatto e la curva passa per il punto di controllo P corrispondente. Alternativamente possiamo esaminare come cambia il supporto. Aumentare la molteplicità significa parimenti diminuire il numero di funzioni $N_{i,m}(\bar{x}) \neq 0$. Un nodo di molteplicità $m_i = m-1$ è definito da una sola B-spline $N_{i,m}(\bar{x}) = 1$. La combinazione convessa di funzioni in quel punto produrrà come risultato proprio P.

Supponiamo quindi di avere una spline $s(x) = \sum_{i=1}^{m+K} c_i N_{i,m}(x)$. Procediamo aggiungendo nodi

[2] Carl R. de Boor. *A Practical Guide to Splines, Revised Edition*. 2003.

coincidenti in un qualche t_i fino a quando la molteplicità in quel punto risulta essere pari a $m-1$.

$$\begin{aligned} s(x) &= \sum_{i=1}^{m+K+1} \bar{c}_i^{[1]} \bar{N}_{i,m}^{[1]}(x) \\ &= \sum_{i=1}^{m+K+2} \bar{c}_i^{[2]} \bar{N}_{i,m}^{[2]}(x) \\ &= \cdots \\ &= \sum_{i=1}^{m+K+(m-1)} \bar{c}_i^{[m-1]} \bar{N}_{i,m}^{[m-1]}(x) \end{aligned}$$

Poiché $\bar{N}_{i,m}^{[m-1]} = 1$ per quanto detto in precedenza, la valutazione di un punto sulla spline è pari a

$$s(x) = \bar{c}_i^{[m-1]}$$

Per trovare tale valore non è necessaria la definizione di alcuna B-spline.
L'algoritmo di De Boor consiste nella geniale osservazione che questo stesso procedimento può essere applicato a qualunque punto $\bar{x} \in [t_l, t_{l+1}] \subset [0,1]$, anche se esso non coincide con un nodo t_i. In ogni punto di cui cerchiamo $s(\bar{x})$ aggiungiamo esattamente $m-1$ nodi coincidenti. Analogamente a quello che succede all'algoritmo di De Casteljau per le curve di Bézier, il punto di valutazione sarà combinazione lineare convessa dei punti di controllo P_i.
Entriamo ora nel dettaglio. Una qualsiasi spline $s_m(x)$ di ordine $m > 1$ può essere anche scritta come combinazione di B-spline di ordine $m-1$ nel seguente modo:

$$s_m(x) = \sum_{i=l-m+1}^{l} c_i N_{i,m}(x) = \sum_{i=l-m+1}^{l} c_i \left(\frac{x - t_i}{t_{i+m-1} - t_i} N_{i,m-1}(x) + \frac{t_{i+m} - x}{t_{i+m} - t_{i+1}} N_{i+1,m-1}(x)\right)$$

Spezziamo le sommatorie e consideriamole individualmente. La prima risulta pari a 0 per $i = l-m+1$ mentre la seconda risulta nulla per $i = l$. Segue quindi:

$$s_m(x) = \sum_{i=l-m+2}^{l} c_i \left(\frac{x - t_i}{t_{i+m-1} - t_i} N_{i,m-1}(x)\right) + \sum_{i=l-m+1}^{l-1} c_i \left(\frac{t_{i+m} - x}{t_{i+m} - t_{i+1}} N_{i+1,m-1}(x)\right)$$

Esprimiamo la seconda sommatoria in modo tale che inizi per $i = l-m+2$:

$$s_m(x) = \sum_{i=l-m+2}^{l} c_i \left(\frac{x - t_i}{t_{i+m-1} - t_i} N_{i,m-1}(x)\right) + \sum_{i=l-m+2}^{l} c_{i-1} \left(\frac{t_{i+m-1} - x}{t_{i+m-1} - t_i} N_{i,m-1}(x)\right)$$

Ci accorgiamo che possiamo raccogliere le sommatorie e che entrambe hanno in comune il fatto di contenere la quantità $N_{i,m-1}(x)$. Risulta pertanto:

$$s_m(x) = \sum_{i=l-m+2}^{l} c_i^{[1]} N_{i,m-1}(x) \qquad \text{con} \quad c_i^{[1]} = \frac{c_i(x - t_i) + c_{i-1}(t_{i+m-1} - x)}{t_{i+m-1} - t_i}$$

Fermiamoci un attimo a considerare cosa abbiamo fatto. Siamo riusciti ad esprimere la curva $s(x)$ come combinazione di $m-1$ B-spline di ordine $m-1$ semplicemente combinando opportunamente tra loro i coefficienti. Generalmente ad ogni passo j dell'algoritmo avremo:

$$s_m(x) = \sum_{i=l-m+1+j}^{l} c_i^{[j]} N_{i,m-j}(x) \qquad \text{con} \quad c_i^{[j]} = \frac{c_i^{[j-1]}(x - t_i) + c_{i-1}^{[j-1]}(t_{i+m-1} - x)}{t_{i+m-1} - t_i}$$

Ci fermiamo quando $j = m-1$, per ottenere:

$$s_m(x) = \sum_{i=l}^{l} c_i^{[m-1]} N_{i,1}(x)$$

Essendo $N_{i,1}(x) = 1$, concludiamo affermando che:

$$s_m(x) = c_l^{[m-1]}$$

1.1.8 Considerazioni sul posizionamento dei nodi

La posizione dei nodi t_i del nostro vettore esteso $\Delta^* = \{t_i\}_{i=1}^{2m+n}$ modifica in maniera sostanziale la forma della curva. Questo motivo è dato dal fatto che vengono modificati gli intervalli sui quali sono definite le varie funzioni B-spline. Scendiamo più nel dettaglio e analizziamo in ordine cosa succede se si spostano i nodi veri oppure i nodi fittizi. Consideriamo per semplicità di avere una curva spline a nodi semplici.

Curve B-spline uniformi e non uniformi

Una curva B-spline definita su una partizione nodale estesa in cui i k nodi veri $t_{m+1}, ..., t_{m+k}$ dividono l'intervallo $[0, 1]$ in $k+1$ parti uguali sono definite *uniformi*. In questo caso la posizione dei punti di controllo non influisce su quella dei nodi veri. In generale, vale la seguente relazione:

$$t_{m+1} = \frac{1}{k+1}$$
$$t_i = \frac{1}{k+1} + t_{i-1} \qquad t = m+2, ..., m+k$$

Vediamo un esempio. Una curva spline con parametrizzazione uniforme definita da 5 punti di controllo possiede un solo nodo vero ($k = 1$) posizionato esattamente a metà dell'intervallo, in quando $t_5 = \frac{1}{2}$. La partizione nodale estesa è la seguente:

$$\Delta^* = \{0, 0, 0, 0, \frac{1}{2}, 1, 1, 1, 1\}$$

Se vogliamo che la posizione dei nodi veri rifletta la posizione dei punti di controllo dobbiamo rendere la nostra partizione nodale *non uniforme*. I modi per farlo sono innumerevoli. Esistono tuttavia alcune parametrizzazioni ben note in letteratura che ho scelto di inserire nel software:

- Parametrizzazione della corda
- Parametrizzazione centripeta

La parametrizzazione della corda tiene in considerazione la distanza fra due punti di controllo consecutivi P_i e P_{i+1}. Tanto più grande sarà il rapporto fra questo valore e la lunghezza totale dell'intera spezzata tanto più esteso sarà, in linea teorica, l'intervallo tra i corrispondenti nodi veri. Ricordiamo tuttavia al lettore che i nodi veri k sono esattamente m in meno degli N punti di controllo (secondo la relazione $N = k + m$), quindi non esisterà una perfetta corrispondenza.
Il metodo per il posizionamento dei nodi veri è il seguente. Si determina prima di tutto la lunghezza totale della spezzata del poligono di controllo L, che sarà utile per normalizzare i valori che troveremo:

$$L = \sum_{i=1}^{N-1} \|P_{i+1} - P_i\|_2 = \sum_{i=1}^{N-1} \sqrt{(x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2}$$

Costruiamo ora un vettore $v = \{v_1, ..., v_N\}$ che accumula le distanze, secondo il seguente procedimento:

$$v_i = \begin{cases} 0 & i = 1 \\ v_{i-1} + \dfrac{\|P_i - P_{i-1}\|_2}{L} & i = 2, ..., N-1 \\ 1 & i = N \end{cases}$$

Ora prendiamo i valori per i nodi veri come segue:

$$t_{m+i} = v_{\lfloor \frac{N-1}{2} \rfloor + i} \qquad i = 1, ..., k$$

Consideriamo questo esempio. Supponiamo di avere 6 punti di controllo tali che $v = \{0, 0.1, 0.3, 0.45, 0.8, 1\}$. I nodi veri risultano essere:

$$t_5 = v_{\lfloor \frac{5}{2} \rfloor + 1} = v_3 = 0.3 \qquad t_6 = v_{\lfloor \frac{5}{2} \rfloor + 2} = v_4 = 0.45$$

Una possibile partizione nodale potrebbe pertanto essere:

$$\Delta^* = \{0, 0, 0, 0, 0.3, 0.45, 1, 1, 1, 1\}$$

La parametrizzazione centripeta è una variante di quella della corda e si ottiene ponendo sotto radice quadrata la distanza tra i punti. Il vettore v diventa:

$$v_i = \begin{cases} 0 & i = 1 \\ v_{i-1} + \dfrac{\sqrt{\|P_i - P_{i-1}\|_2}}{L} & i = 2, ..., N-1 \\ 1 & i = N \end{cases}$$

e il procedimento rimane invariato.

Curve B-spline chiuse, aperte e interpolanti agli estremi

Questo paragrafo riguarderà il posizionamento dei nodi fittizi e di come essi contribuiscano a generare tipi diversi di spline. Ricordiamo che una partizione nodale estesa prevede l'inserimento di $m-1$ nodi ≤ 0 a sinistra e $m-1$ nodi ≥ 1 a destra.
Consideriamo il caso più semplice, visto in tutti gli esempi fatti finora. Se tali nodi vengono scelti coincidenti con gli estremi, tale curva viene detta *interpolante agli estremi* (*clamped* in inglese). Questo significa che la curva avrà origine nel primo punto di controllo e avrà termine esattamente nell'ultimo. Il motivo per cui questo accade è già stato ampiamente descritto nelle due sottosezioni precedenti: aggiungendo tali nodi coincidenti aumentiamo la molteplicità in quel singolo punto e la forziamo la curva ad interpolarlo. La partizione avrà questa forma:

$$\Delta^* = 0, 0, 0, 0, t_{m+1}, ..., t_{m+k}, 1, 1, 1, 1$$

Molto interessanti sono le curve aperte. Queste - come il nome suggerisce - non interpolano agli estremi la curva. Ne consegue che i nodi fittizi non saranno sicuramente coincidente con gli estremi. La seguente partizione nodale estesa determina curve spline aperte e uniformi:

$$\Delta^* = \{-3, -2, -1, 0, 1, 2, 3, 4\}$$

La seguente determina invece curve aperte non uniformi:

$$\Delta^* = \{-1.3, -1, -0.3, 0, 0.7, 1, 1.7, 2, 2.7\}$$

La partizione estesa periodica, utilizzata per creare curve chiuse e periodiche, si definisce da una partizione uniforme o non uniforme ripetendo gli stessi m intervalli nodali a sinistra e a destra.
Le curve B-spline chiuse sono un'ulteriore evoluzione. Esse prevedono che la curva abbia lo stesso valore nel primo e nell'ultimo punto. Si chiede quindi, formalmente, che $C(0) = C(1)$. Per farlo si è soliti vedere tali curve come concatenazione di esattamente m curve aperte. Se con m punti di controllo $P_1, ..., P_m$ siamo riusciti a definirne una, pare logico pensare di aver bisogno di almeno altri $m-1$ punti P_{m+1}, P_{m+2} e P_{m+3} per le altre. Resta da decidere la posizione in cui inserirli nello spazio. Possiamo intuire che la soluzione risiede nel porre tali punti coincidenti con i primi. In generale, dati k punti già inseriti:

$$P_{k+i} = P_i \qquad i = 1, ..., m-1$$

Così facendo è come se creassimo m curve che sono definite agli estremi sugli stessi intervalli internodali. Così siamo sicuri di legare la fine di una con l'inizio di quella successiva. Tutto ciò di cui abbiamo bisogno è di avere $2m-1$ punti. Una possibile partizione estesa di una curva B-spline chiusa uniforme con 7 punti di controllo è la seguente:

$$\Delta^* = \{-0.75, -0.5, -0.25, 0, 0.25, 0.5, 0.75, 1, 1.25, 1.5, 1.75\}$$

La determinazione della posizione dei nodi segue la formula vista in precedenza per le curve aperte.

1.1.9 Derivata di una curva B-spline

In questa sezione espongo la formula generale per calcolare la derivata di una curva B-spline. Come possiamo ben immaginare esse saranno combinazioni lineari di derivate di funzioni B-spline. Data la curva $C(x) = \sum_{i=l-m+1}^{l} P_i N_{i,m}(x)$, la sua derivata nel punto $t \in [t_l, t_{l+1}]$ è definita come:

$$C(t) = \sum_{i=l-m+1}^{l} P_i N'_{i,m}(t)$$

dove

$$N'_{i,m}(t) = \frac{m-1}{t_{i+m-1} - t_i} N_{i,m-1}(t) - \frac{m-1}{t_{i+m} - t_{i+1}} N_{i+1,m-1}(t)$$

Questo risultato si raggiunge solitamente tramite induzione. La dimostrazione è particolarmente tediosa quindi scelgo di evitarla.
Possiamo subito vedere che la derivata di una B-spline di ordine m è combinazione convessa di due B-spline di ordine $m-1$. Questo ci permette di affermare che esiste sicuramente una forma

alternativa per la derivata prima di tale curva che ci permette di definirla in termini di funzioni B-spline di ordine inferiore. Infatti possiamo scrivere $C(t)$ come:

$$C'(t) = \sum_{i=l-m+1}^{l} P_i(\frac{m-1}{t_{i+m-1}-t_i}N_{i,m-1}(t) - \frac{m-1}{t_{i+m}-t_{i+1}}N_{i+1,m-1}(t))$$

Questa formulazione può essere semplificata ulteriormente in maniera molto simile a quanto fatto in precedenza durante la dimostrazione dell'algoritmo di De Boor. Spezzando le sommatorie otteniamo:

$$C'(t) = (m-1)\sum_{i=l-m+1}^{l} P_i\frac{N_{i,m-1}(t)}{t_{i+m-1}-t_i} - (m-1)\sum_{i=l-m+1}^{l} P_i\frac{N_{i+1,m-1}(t)}{t_{i+m}-t_{i+1}}$$

Esprimiamo la prima sommatoria come se l'indice i partisse da $l-m$. Equivalentemente:

$$C'(t) = (m-1)\sum_{i=l-m}^{l-1} P_{i+1}\frac{N_{i+1,m-1}(t)}{t_{i+m}-t_{i+1}} - (m-1)\sum_{i=l-m+1}^{l} P_i\frac{N_{i+1,m-1}(t)}{t_{i+m}-t_{i+1}}$$

In queste sommatorie almeno una B-spline risulterà pari a 0. Per la precisione questo avviene per $i = l-m$ per la prima e per $i = l$ per la seconda. Possiamo quindi scrivere:

$$C'(t) = (m-1)\sum_{i=l-m+1}^{l-1} P_{i+1}\frac{N_{i+1,m-1}(t)}{t_{i+m}-t_{i+1}} - (m-1)\sum_{i=l-m+1}^{l-1} P_i\frac{N_{i+1,m-1}(t)}{t_{i+m}-t_{i+1}}$$

Posso ora raccogliere: la quantità $(m-1)\frac{N_{i+1,m-1}(t)}{t_{i+m}-t_{i+1}}$ compare in entrambe.

$$C'(t) = \sum_{i=l-m+1}^{l-1} \frac{N_{i+1,m-1}(t)}{t_{i+m}-t_{i+1}}(m-1)(P_{i+1}-P_i)$$

Definendo

$$Q_i = (m-1)\frac{P_{i+1}-P_i}{t_{i+m}-t_{i+1}}$$

Arriviamo a definire la derivata prima della curva come:

$$C'(t) = \sum_{i=l-m+1}^{l-1} Q_i N_{i+1,m-1}(t)$$

È quindi possibile esprime tale curva come combinazione lineare convessa di $m-1$ funzione B-spline di ordine $m-1$, i cui punti di controllo Q_i sono a loro volta dati dalla combinazione lineare dei punti originali (P_{i+1} e P_i).

1.2 Interpolazione

1.2.1 Interpolazione tramite B-spline

Il problema dell'interpolazione può essere risolto facendo uso delle funzioni B-spline. Consideriamo un insiemi di punti (x_i, y_i), $i = 0, ..., N+1$ che desideriamo interpolare. Data una partizione di nodi $\Delta = \{a < t_0 < t_1 < ... < t_k < t_{k+1}\}$, si vuole calcolare la funzione $s(x) \in S_m(x)$ tale che $s(x_i) = y_i$, $i = 0, ..., N+1$. Tale funzione $s(x)$ dipende fortemente dalla posizione dei k nodi veri. Come già discusso, per risolvere un problema di questo tipo abbiamo necessariamente bisogno di un numero di condizioni pari alla dimensione dello spazio di riferimento. Vale pertanto che $N+2 = k+m$. Di conseguenza $S_m(\Delta)$ dovrà avere $N+2$ gradi di libertà e esattamente $k = N-m+2$ nodi veri. Il sistema che viene generato è il seguente:

$$\begin{cases} N_{1,m}(x_0)\alpha_1 + N_{2,m}(x_0)\alpha_2 + \cdots + N_{N+2,m}(x_0)\alpha_{N+2} = y_0 \\ N_{1,m}(x_1)\alpha_1 + N_{2,m}(x_1)\alpha_2 + \cdots + N_{N+2,m}(x_1)\alpha_{N+2} = y_1 \\ \vdots \\ N_{1,m}(x_{N+1})\alpha_1 + N_{2,m}(x_{N+1})\alpha_2 + \cdots + N_{N+2,m}(x_{N+1})\alpha_{N+2} = y_{N+1} \end{cases}$$

Questo sistema è quadrato ma non possiamo ancora essere certi riguardo la sua effettiva risoluzione. Potrebbe infatti capitare che il rango non sia massimo, in quanto certe scelte dei nodi veri potrebbero generare colonne con elementi tutti pari a 0.
Il teorema di Schoenberg-Whitney definisce una condizione necessaria e sufficiente affinché questo non avvenga e il sistema abbia sempre un'unica soluzione.

Teorema di Schoenberg-Whitney. *Sia definita una successione crescente di dati da interpolare* $(\xi_i, f(\xi_i))$, $i = 1, ..., s$. *Siano inoltre definiti una successione crescente di nodi* $\Delta = \{t_1 < ... < t_{s+m}\}$, *il relativo spazio* $S_m(\Delta)$ *e la relativa successione delle B-spline* $N_{1,m}(x), ..., N_{s,m}(x)$.
Il problema può essere espresso come il seguente sistema lineare:

$$\sum_{j=1}^{s} \alpha_j N_{j,m}(\xi_i) = f(\xi_i) \qquad i = 1, .., s$$

con matrice quadrata A di dimensioni sxs tale che:

$$A = \begin{bmatrix} N_{1,m}(\xi_1) & N_{2,m}(\xi_1) & \cdots & N_{s,m}(\xi_1) \\ N_{1,m}(\xi_2) & N_{2,m}(\xi_2) & \cdots & N_{s,m}(\xi_2) \\ \vdots & \vdots & \ddots & \vdots \\ N_{1,m}(\xi_s) & N_{2,m}(\xi_s) & \cdots & N_{s,m}(\xi_s) \end{bmatrix}$$

La matrice A ha rango massimo se e solo se $t_i < \xi_i < t_{i+m}$, $i = 1, ..., s$.

Questo si verifica se per ciascuna B-spline esiste almeno un punto di interpolazione nel suo supporto.
Shoenberg stesso ci fornisce una opportuna scelta dei nodi:

$$k = N - 2 \qquad t_0 \equiv x_0, \quad t_1 \equiv x_2, \quad ... \quad t_n \equiv x_{N-1}, \quad t_{N+1} \equiv x_{k+1}$$

Un'altra soluzione interessante consiste nello scegliere $k = N$ nodi veri e porre $t_i \equiv \xi i$, $i = 1, ..., N$. Così facendo si ottengono esattamente $N+2$ equazioni con $N+m$ incognite. Sarà quindi necessario aggiungere $m - 2$ condizioni ai bordi, che possono essere scelte nel seguente modo.

1.2.2 Condizioni al contorno

Le due condizioni aggiuntive possono essere scelte in vari modi. Si è solito inserirle agli estremi della curva, in maniera tale da non modificare in maniera sostanziale la parte interna di essa. Questo ulteriore concetto verrà approfondito quando si parlerà di supporto locale. Esse vengono chiamate *al contorno* in quanto interessano direttamente il primo e l'ultimo nodo della curva.
Di seguito elenchiamo le più diffuse:

1. $s'_m(x_0) = y'_0, \qquad s'_m(x_{n+1}) = y'_{n+1}$
2. $s''_m(x_0) = 0, \qquad s''_m(x_{n+1}) = 0$
3. $s'_m(x_0) = s'_m(x_{n+1}), \qquad s''_m(x_0) = s''_m(x_{n+1}) \qquad$ se $y_0 \equiv y_{n+1}$

La seconda e la terza condizione determinano una spline che viene solitamente definita come, rispettivamente, *naturale* oppure *periodica*. Le curve spline cubiche naturali sono particolarmente adatte tra l'insieme delle curve interpolanti in quanti minimizzano la seguente quantità:

$$\int_{x_0}^{x_{n+1}} (s''_m(x))^2 dx$$

Questo garantisce che la curva sia la meno oscillante possibile.
A questo punto abbiamo tutto ciò che ci serve per determinare univocamente la spline. Sarà sufficiente risolvere un sistema lineare per trovare il valore di tutti i coefficienti. Notare bene che se prendiamo funzioni di basi diverse da quelle polinomiali, come succede nel caso delle B-spline, la matrice derivata dal sistema potrebbe variare.

1.3 Approssimazione ai minimi quadrati

Non sempre l'interpolazione potrebbe fare al caso nostro. Se il numero di dati da interpolare è troppo alto c'è la concreta possibilità che la loro posizione forzi la curva ad avere una forma per nulla soddisfacente. Il risultato potrebbe essere pertanto poco significativo o portarci a conclusioni sbagliate. Per questo motivo quando questo si verifica si è soliti cambiare strategia ed accontentarsi di un'approssimazione dei dati. La curva approssimante non passerà necessariamente per tali punti ma cercherà il più possibile di avvicinarsene.
Formuliamo il problema in maniera più generale.
Consideriamo un insieme di punti da approssimare (x_i, y_i), $i = 1, ..., m$. Cerchiamo una funzione $f(x)$ che minimizzi la distanza fra se stessa e i dati forniti. La distanza tra un punto e la sua approssimazione viene solitamente definita *errore* e vale la seguente relazione:

$$e_i = f_n(x_i) - y_i \qquad i = 0, ..., m$$

Indicando con e_i gli errori di approssimazione, osserviamo subito che non abbiamo alcuna garanzia sul fatto che questi abbiano tutti lo stesso segno. Per evitare che gli errori si cancellino l'un con l'altro si richiede solitamente di minimizzare la somma dei quadrati degli errori. Se definiamo tale grandezza con S, risulta che:

$$S = \sum_{i=1}^{m} e_i^2$$

Un modo alternativo per definire la distanza è servirsi della norma euclidea:

$$\|y - f_n\|_2 = \sqrt{\sum_{i=1}^{m} (y_i - f_n(x_i))^2}$$

Assegnati i punti da approssimare (x_i, y_i), $i = 1, ..., m$ e scelte $n + 1$ funzioni base Φ_j, $j = 0, ..., n$ (con $n < m$) si vuole trovare la funzione approssimante $f_n(x) = a_0\Phi_0 + ... + a_n\Phi n = \sum_{j=0}^{n} a_j\Phi_j$ di grado n in maniera tale che sia minimo lo scarto quadratico medio S. Questo si riduce a determinare i coefficienti a_i, $i = 0, .., n$ tali per cui:

$$\min S = \min_{a_0,...,a_n} \|y - f_n(x)\|_2^2 = \min_{a_0,...,a_n} \sum_{i=1}^{m} (y_i - f_n(x_i))^2$$

Dal punto di vista matriciale possiamo riscrivere $f_n(x)$ come Ha, dove H è la matrice delle funzioni basi ($H|_{ij} = \phi_j(x_i)$). Di conseguenza:

$$y - Ha = \begin{bmatrix} y_1 \\ y_2 \\ \cdots \\ \cdots \\ \cdots \\ y_m \end{bmatrix} - \begin{bmatrix} \phi_0(x_1) & \phi_1(x_1) & \cdots & \phi_n(x_1) \\ \phi_0(x_2) & \phi_1(x_2) & \cdots & \phi_n(x_2) \\ \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ \phi_0(x_m) & \phi_1(x_m) & \cdots & \phi_n(x_m) \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ \cdots \\ a_n \end{bmatrix}$$

Poiché il nostro obiettivo è trovare i coefficienti a_i per cui suddetta quantità è minima, il problema si riduce alla risoluzione del sistema lineare

$$Ha = y$$

Se $m > n$ il sistema è sovradeterminato e può portare ad una soluzione approssimante. Se invece $m = n$ allora il problema si riconduce all'interpolazione.
Esistono diversi modi in letteratura per risolvere il sistema nel senso dei minimi quadrati e ne citiamo i più famosi:

- Metodo delle equazioni normali
- Metodo QR-LS
- Metodo della decomposizioni in valori singolari (SVD-LS)

1.3.1 Metodo delle equazioni normali (caso polinomiale)

Consideriamo per comodità una funzione polinomiale approssimante $f_n(x) = a_0 + a_1x + ... + a_nx^n = \sum_{i=0}^{n} a_i x^i$.
La quantità S e la matrice H diventano:

$$S = \sum_{j=1}^{m} (y_j - \sum_{i=0}^{n} a_i x_j^i)^2 \qquad H = \begin{bmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^n \\ 1 & x_2 & x_2^2 & \cdots & x_2^n \\ 1 & x_3 & x_3^2 & \cdots & x_3^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_m & x_m^2 & \cdots & x_m^n \end{bmatrix}$$

La matrice H è un esempio di matrice di Vandermonde ed è particolarmente mal condizionata. Il nostro obiettivo è quello di definire univocamente i coefficienti a_i. La funzione S è multivariata

e per trovarne il minimo dobbiamo considerare le sue derivate parziali e porle tutte pari a 0. Il sistema che cerchiamo di risolvere è il seguente:

$$\begin{cases} \dfrac{dS}{da_0} = 0 \\ \dfrac{dS}{da_1} = 0 \\ \vdots \\ \dfrac{dS}{da_n} = 0 \end{cases}$$

Prima di proseguire con la soluzione ritengo utile veder due esempi pratici di calcoli con funzioni di grado basso.

Poniamo $n = 1$ e cerchiamo di conseguenza la retta che approssima ai minimi quadrati i punti (x_i, y_i), $i = 1, ..., m$. Considerando $\sum_{j=1}^{m}(y_j - \sum_{i=0}^{1} a_i x_j^i)^2 = \sum_{j=1}^{m}(y_j - (a_0 + a_1 x))^2$ il sistema diventa:

$$\begin{cases} \dfrac{dS}{da_0} = 0 \\ \dfrac{dS}{da_1} = 0 \end{cases}$$

Calcoliamo ora le derivate parziali.

$$\frac{dS}{da_0} = \sum_{j=1}^{m} 2(y_j - a_0 - a_1 x_j)(-1) = 2(-\sum_{j=1}^{m} y_j + \sum_{j=1}^{m} a_0 + \sum_{j=1}^{m} a_1 x_j)$$

$$\frac{dS}{da_1} = \sum_{j=1}^{m} 2(y_j - a_0 - a_1 x_j)(-x_j) = 2(-\sum_{j=1}^{m} x_j y_j + \sum_{j=1}^{m} a_0 x_j + \sum_{j=1}^{m} a_1 x_j^2)$$

Poniamole pari a 0 e risolviamo il sistema.

$$\begin{cases} \dfrac{dS}{da_0} = 0 \\ \dfrac{dS}{da_1} = 0 \end{cases} = \begin{cases} 2(-\sum_{j=1}^{m} y_j + \sum_{j=1}^{m} a_0 + \sum_{j=1}^{m} a_1 x_j) = 0 \\ 2(-\sum_{j=1}^{m} x_j y_j + \sum_{j=1}^{m} a_0 x_j + \sum_{j=1}^{m} a_1 x_j^2) = 0 \end{cases} = \begin{cases} \sum_{j=1}^{m} a_0 + \sum_{j=1}^{m} a_1 x_j = \sum_{j=1}^{m} y_j \\ \sum_{j=1}^{m} a_0 x_j + \sum_{j=1}^{m} a_1 x_j^2 = \sum_{j=1}^{m} x_j y_j \end{cases}$$

Scriviamo il tutto in forma matriciale.

$$\begin{bmatrix} m & \sum_{j=1}^{m} x_j \\ \sum_{j=1}^{m} x_j & \sum_{j=1}^{m} x_j^2 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \end{bmatrix} = \begin{bmatrix} \sum_{j=1}^{m} y_j \\ \sum_{j=1}^{m} x_j y_j \end{bmatrix}$$

Il sistema sopra descritto è definito *sistema delle equazioni normali* e la risoluzione di questo tramite un qualsiasi metodo (diretto o indiretto) porta alla definizione dei coefficienti a_0 e a_1.
Consideriamo ora il caso con $n = 2$. Procediamo analogamente ponendo a 0 le tre derivate parziali. Il sistema relativo, in forma matriciale, è il seguente:

$$\begin{bmatrix} m & \sum_{j=1}^{m} x_j & \sum_{j=1}^{m} x_j^2 \\ \sum_{j=1}^{m} x_j & \sum_{j=1}^{m} x_j^2 & \sum_{j=1}^{m} x_j^3 \\ \sum_{j=1}^{m} x_j^2 & \sum_{j=1}^{m} x_j^3 & \sum_{j=1}^{m} x_j^4 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} \sum_{j=1}^{m} y_j \\ \sum_{j=1}^{m} x_j y_j \\ \sum_{j=1}^{m} x_j^2 y_j \end{bmatrix}$$

Notiamo che sembra esserci uno schema ben preciso. La matrice quadrata - che si denota solitamente con G - è simmetrica e i suoi elementi sono potenze di sommatorie. Queste potenze crescono all'aumentare della riga e della colonna. La stessa cosa si può vedere nei confronti del secondo membro dell'equazione, che chiamiamo b.
Esaminiamo ora il caso generale. Dato un qualsiasi grado n (con $n < m$) il sistema lineare di ordine $n + 1$ che dobbiamo risolvere per trovare i coefficienti a_i, $i = 0, ..., n$ della funzione approssimante $f_n(x)$ è il seguente:

$$\begin{bmatrix} m & \sum_{j=1}^{m} x_j & \cdots & \sum_{j=1}^{m} x_j^n \\ \sum_{j=1}^{m} x_j & \sum_{j=1}^{m} x_j^2 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ \sum_{j=1}^{m} x_j^n & \cdots & \cdots & \sum_{j=1}^{m} x_j^{2n} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} \sum_{j=1}^{m} y_j \\ \sum_{j=1}^{m} x_j y_j \\ \vdots \\ \sum_{j=1}^{m} x_j^n y_j \end{bmatrix}$$

L'efficacia di questo metodo risiede nel fatto che sia G che b possono essere fattorizzati in maniera tale che uno dei due fattori sia proprio la matrice H definita in partenza. Risulta infatti vero che $G = H^\mathsf{T} H$ e $b = H^\mathsf{T} y$, come mostrato di seguito:

$$\begin{bmatrix} 1 & 1 & \cdots & \cdots & 1 \\ x_1 & x_2 & \cdots & \cdots & x_m \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ x_1^n & x_2^n & \cdots & \cdots & x_m^n \end{bmatrix} \begin{bmatrix} 1 & x_1 & \cdots & x_1^n \\ 1 & x_2 & \cdots & x_2^n \\ \vdots & \vdots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_m & \cdots & x_m^n \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} 1 & 1 & \cdots & \cdots & 1 \\ x_1 & x_2 & \cdots & \cdots & x_m \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ x_1^n & x_2^n & \cdots & \cdots & x_m^n \end{bmatrix} \begin{bmatrix} y_1 \\ y_1 \\ \vdots \\ \vdots \\ y_m \end{bmatrix}$$

Definita pertanto opportunamente la matrice H, possiamo risolvere il sistema $H^\mathsf{T} H a = H^\mathsf{T} y$ (*sistema delle equazioni normali*) e avere la garanzia che la soluzione soddisfi il problema minimo di

partenza. In altre parole:

$$\min S = \|y - f_n(x)\|_2^2 \Leftrightarrow H^\mathsf{T} H a = H^\mathsf{T} y$$

Il problema dell'approssimazione ai minimi quadrati ha sempre soluzione. Questa è unica se H ha rango massimo e di conseguenza se $H^\mathsf{T} H$ è non singolare. In caso contrario le soluzioni sono infinite, ma quella di lunghezza minima rimane unica.

1.3.2 Approssimazione ai minimi quadrati (caso B-Spline)

La matrice di Vandermonde ha il vantaggio di essere facilmente calcolabile ma la sua più grossa pecca è il fatto di essere particolarmente mal condizionata. Questo significa che all'aumentare dei punti (x_i, y_i) da approssimare l'algoritmo di risoluzione dei sistemi proposti nella sottosezione precedente risulta poco stabile e questo determina un risultato che diventa sempre più impreciso. Un opportuno cambio della base delle funzioni potrebbe essere la soluzione giusta. Intendo muovermi verso questa direzione mostrando come la base delle B-spline sia particolarmente adatta.
Consideriamo un insieme di punti (x_i, y_i), $i = 1, .., N$ da approssimare e stabiliamo l'ordine m delle B-spline. Si è soliti definire uno spazio delle spline $S_m(\Delta^*)$ di dimensioni $m + k = \frac{N}{2}$, ovvero pari a circa la metà dei punti dati. Il vettore di nodi esteso $\Delta^* = \{t_{-m+1} \leq ... \leq a = t_0 < t_1 < ... < t_k < t_{k+1} = b \leq ... \leq t_{k+m}\}$ avrà pertanto $k = \frac{N}{2} - m$ nodi veri. La funzione che si ottiene sostituendo ad una base generica quella delle B-spline è la seguente:

$$f(x) = a_1 N_{1,m}(x) + ... + a_{m+k} N_{m+k,m}(x) = \sum_{j=1}^{m+k} a_j N_{j,m}$$

Il procedimento ora risulta simile a quello visto in precedenza. Il nostro obiettivo è sempre quello di determinare i coefficienti a_i, $i = 1, ..., m + k$ in maniera tale da avere una curva risultante che approssima ai minimi quadrati i punti dati. Abbiamo già discusso del fatto che ciò equivale a risolvere il sistema lineare $H^\mathsf{T} H a = H^\mathsf{T} y$ ma questa volta le matrici H e la sua trasposta H^T vengono definite in questo modo:

$$H = \begin{bmatrix} N_{1,m}(x_1) & N_{2,m}(x_1) & \cdots & N_{m+k,m}(x_1) \\ N_{1,m}(x_2) & N_{2,m}(x_2) & \cdots & N_{m+k,m}(x_2) \\ \vdots & \vdots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ N_{1,m}(x_N) & N_{2,m}(x_N) & \cdots & N_{m+k,m}(x_N) \end{bmatrix}$$

$$H^{\mathsf{T}} = \begin{bmatrix} N_{1,m}(x_1) & N_{1,m}(x_2) & \cdots & \cdots & \cdots & N_{1,m}(x_N) \\ N_{2,m}(x_1) & N_{2,m}(x_2) & \cdots & \cdots & \cdots & N_{2,m}(x_N) \\ \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ N_{m+k,m}(x_1) & N_{m+k,m}(x_2) & \cdots & \cdots & \cdots & N_{m+k,m}(x_N) \end{bmatrix}$$

Dobbiamo tenere sempre a mente che le B-spline $N_{i,m}(x)$ dovranno sempre essere valutate nell'intervallo opportuno. Al momento della creazione della matrice H dobbiamo, per ogni punto x_j, stabilire qual è il suo supporto su Δ^* e qui valutare la funzione.
I motivi per il quale scegliere di utilizzare una base di B-spline sono molteplici. Di seguito discuto dei più importanti.

Vantaggi nell'utilizzo di B-spline per l'approssimazione ai minimi quadrati

Il vantaggio principale dell'utilizzo di tali funzioni consiste nella possibilità di poter adattare l'algoritmo di De Boor per la valutazione della curva tramite i coefficienti trovati. Determinati i valori di a_i, $i = 1, ..., m + k$, possiamo applicare il suddetto algoritmo sostituendo ai punti di controllo i coefficienti. Questo è dimostrabile intuitivamente: dato $\bar{x} \in [t_l, t_{l+1}]$, se il procedimento permette di valutare rapidamente la funzione $\sum_{i=l-m+1}^{l} c_i N_{i,m}(x)$ (dove c_i sono i punti di controllo) allora potrà certamente fare lo stesso della funzione $\sum_{i=l-m+1}^{l} a_i N_{i,m}(x)$.
Il secondo vantaggio notevole che subito possiamo apprezzare è il fatto che ogni funzione B-spline ha valore compreso tra 0 e 1. Questo ha due conseguenze: le matrici H e H^{T} sono composte da solo valori positivi e questi sono sufficientemente piccoli da contribuire ad un ottimo indice di condizionamento. Infatti la condizione $0 \leq N_{i,m}(x) \leq 1$ risulta vera anche con valori molto alti di m.
Il terzo, conseguenza immediata del precedente, riguarda la matrice $G = H^{\mathsf{T}}H$. Questa matrice è sicuramente simmetrica, definita positiva e soprattutto a diagonale dominante e a bande. Questa informazione facilita enormemente la risoluzione del sistema lineare $Ga = b$, dove $b = H^{\mathsf{T}}y$. Infatti tali le condizioni sono rispettate possiamo servirci della decomposizione (o fattorizzazione) di Cholesky per scrivere G come il prodotto di una matrice triangolare inferiore R e della sua trasposta R^{T}. Il teorema ci garantisce l'unicità e l'esistenza di questa nuova matrice. Dato quindi $G = RR^{\mathsf{T}}$ il nuovo sistema diventa:

$$RR^{\mathsf{T}}a = b$$

Ponendo $R^{\mathsf{T}}a = z$ possiamo risolvere l'equivalente sistema lineare:

$$\begin{cases} Rz = b \\ R^{\mathsf{T}}a = z \end{cases}$$

Essendo le matrici triangolari, la complessità computazione di questo sistema si riduce ulteriormente. L'applicazione dei metodi di sostituzione in avanti (*forward substitution*) o all' indietro (*backward substitution*) determinano infine i coefficienti.
IL software si serve dell'implementazione della decomposizione di Cholesky fornito da GSL tramite le funzioni `gsl_linalg_cholesky_decomp` e `gsl_linalg_cholesky_solve`.

Capitolo 2

Controllo delle animazioni lungo un percorso

La nostra animazione è un esempio di *camera motion* ovvero simula il movimento di una telecamera all'interno di una scena. Essa si muove lungo un percorso (*camera path*) ad una certa velocità orientandosi verso alcuni obiettivi. Il software permette all'utente di controllare ogni singolo aspetto di tale movimento e le sezioni che seguono descrivono nel dettaglio ognuno di essi.

2.1 Camera path

Il nostro percorso è una curva spline $C(t)$ di ordine m in tre dimensioni. Per poterla visualizzare è pertanto necessario che l'utente definisca un poligono di controllo di almeno m punti. Ad ogni frame, in base alla velocità richiesta, la telecamera viene fatta avanzare oppure indietreggiare lungo il percorso e ne viene valutato l'orientamento. Come spiegato in maniera più dettagliata di seguito, sarà necessario che tale curva sia parametrizzata alla lunghezza d'arco. La forma di tale curva è a completa discrezione dell'utente, che può decidere non solo la posizione dei punti di controllo ma anche la parametrizzazione della partizione nodale (alla corda o centripeta) o il tipo stesso di traiettoria (aperta, chiusa o interpolante agli estremi).

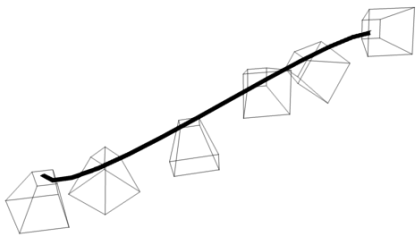


Figura 2.1: Esempio di camera path con la telecamera in vari istanti dell'animazione

2.2 Velocità

2.2.1 Considerazioni sul tempo d'animazione

La telecamera si muove lungo la curva ad una certa velocità. Tale velocità può essere o non essere uniforme lungo il percorso. La più basilare legge del moto ci dice che c'è un rapporto che lega velocità, spazio e tempo:

$$\bar{v}(t) = \frac{dS(t)}{dt}$$

Possiamo vedere tale grandezza come la quantità di spazio percorso in una certa quantità di tempo, ovvero come la derivata della funzione $S(t)$. Facciamo una piccola premessa relativa al tempo e come esso viene misurato.

Prendendo in prestito il lessico cinematografico, una animazione di qualsiasi tipo è composta di una serie di fotogrammi (*frames*) che vengono mostrati in rapida successione, dando il senso del movimento. Tali fotogrammi sono come delle istantanee che catturano la scena in un certo momento. La velocità è misurata solitamente in termini di *frames per second* (*fps*), ovvero il numero di fotogrammi che vengono alternati in un secondo. Un esempio è opportuno. Immaginiamo di voler creare un video a partire da 500 fotografie. Se impostiamo una velocità pari a 25 fps, ogni secondo verranno alternate 25 fotografie e la durata totale del video sarà pari a $t_1 = \frac{500}{25} = 20s$. Se invece impostassimo una velocità pari a 50 fps allora il video durerebbe la metà, in quanto avremo $t_2 = \frac{500}{50} = 10s$.

La velocità di animazione viene solitamente mantenuta constante. Questo significa che la durata finale di un'animazione sarà direttamente proporzionale al numero di frame. Il software prevede una camera motion di una telecamera che si muove su una curva valutata in n punti. Poniamo

come condizione necessaria che dopo esattamente n frame la camera sia passata dal punto iniziale al punto finale. Questo significa che il tempo necessario per percorrere interamente la curva è anch'esso costante. Definito con F il numero di frame al secondo, abbiamo un tempo totali di animazione

$$t_{tot} = \frac{n}{F}$$

2.2.2 Parametrizzazione alla lunghezza d'arco

La condizione posta precedentemente non dice nulla riguarda alla velocità con la quale la curva viene percorsa. Sappiamo soltanto il tempo totale di percorrenza. La velocità varia quindi da frame in frame. Per poter controllare la velocità è pertanto necessario attuare una parametrizzazione in maniera tale che la curva possa essere valutata su dei punti che la partizionano in intervalli di uguale lunghezza. Questa è detta parametrizzazione *alla lunghezza d'arco*. Consideriamo pertanto una curva $C(t) : [0, 1] \to R^n$. La funzione lunghezza ad arco associata alla curva $C(t)$ che definisce lo spazio percorso su di essa al tempo t è la seguente:

$$L(t) = \int_0^t \|C'(t)\|_2 dt$$

La velocità è geometricamente intesa come la derivata dello spazio:

$$v(t) = L'(t) = \|C'(t)\|_2$$

$v(t)$ è sempre positiva. Questo ci permette di dire che $L(t)$ è una funzione monotona, crescente e invertibile. $C(t)$ può pertanto essere riparametrizzata ed esiste sempre una parametrizzazione che garantisca una velocità scalare costante unitaria $v(t) = 1$ in ogni intervallo.
Consideriamo infatti una curva $C2 = C(L^{-1}(t))$. $C2$ è equivalente a C. Inoltre:

$$C2'(t) = C'(L^{-1}(t))(L^{-1}(t)) = \frac{C'(L^{-1}(t))}{\|C'(L^{-1}(t))\|_2}$$

e di conseguenza

$$\|C2'(t)\|_2 = 1$$

Abbiamo pertanto trovato la parametrizzazione che fa al caso nostro, in quanto la nuova curva $C2$ è equivalente a quella di partenza ma la sua velocità di percorrenza è sempre costante e pari ad 1. Vale pertanto che

$$L2(t) = \int_0^t \|C2'(t)\|_2 dt = \int_0^t dt = t$$

Al valore t del parametro corrisponde esattamente una lunghezza t sulla curva.

2.2.3 Funzione inversa dello spazio e *look-up table*

La lunghezza della curva viene solitamente considerata normalizzata. $C(t)$ è definita per valori di $t \in [0,1]$ e pertanto è sufficiente dividere la lunghezza di ogni tratto di curva per $L(1)$.
Abbiamo visto che per parametrizzare una curva alla lunghezza d'arco abbiamo bisogno di calcolare la funzione inversa dello spazio. Il modo più rapido di calcolare $L(t)$ è senza dubbio sommare le distanze tra i punti valutati P_i. Così facendo approssimiamo la lunghezza della curva con la lunghezza della spezzata, come si può vedere al punto 1.1.1. Tale funzione è detta *chord length*. Dati n punti $P_j = C(t_j)$, $j = 1, ..., n$ valutati sulla curva, tale funzione permette di approssimare $L(t_i)$ in questo modo:

$$L(t_i) = \sum_{j=1}^{i} \|P_j - P_{j-1}\|_2$$

L'approssimazione risulta sempre più simile alla lunghezza esatta al crescere dei punti P_i. Possiamo quindi creare una tabella, che prende il nome di *look-up table*, contenente per tutti gli n punti di valutazione della curva equispaziati $t \in [0,1]$ il valore $L(t) \in [0,1]$ corrispondente. Un esempio potrebbe essere il seguente:

Tabella 2.1: Look-up table 1

frame i	t_i	$L(t_i)$
1	0	0
2	0.0025	0.005
3	0.0050	0.0081
4	0.0075	0.0095
5	0.01	0.0108
⋮	⋮	⋮
⋮	⋮	⋮
$n-2$	0.95	0.91
$n-1$	0.975	0.97
n	1	1

Osserviamo immediatamente che la funzione $L(t_i)$ non cresce in maniera costante all'incremento del valore i.
Conosciamo però come dovrebbe essere per avere una velocità costante. Dati n punti di valutazione, deve risultare vera la seguente relazione:

$$L(\hat{t_i}) = \frac{i-1}{n-1} \qquad i = 1, ..., n \tag{2.1}$$

dove $\hat{t_i}$ sono i nuovi punti di valutazione. Una tabella che rappresenta una curva parametrizzata all'arco - quindi a velocità di percorrenza costante lungo la curva - dovrebbe essere circa come segue (con $n = 400$, ad esempio):

Tabella 2.2: Look-up table 2

frame i	$\hat{t_i}$	$L(\hat{t_i})$
1	0	0
2	?	0.0025
3	?	0.005
4	?	0.0075
5	?	0.01
⋮	⋮	⋮
⋮	⋮	⋮
398	?	0.995
399	?	0.9975
400	1	1

Una volta assegnata la curva percorso e costruita la tabella che in generale si presenta come la tabella 2.1, per poter avere andamento lungo il percorso con velocità costante si procede come segue:

1. Per ogni istante (frame) equispaziato $i = 1, ..., n$ calcoliamo la distanza percorsa a tale istante con incremento costante
$$s_i = \frac{i-1}{n-1}$$
2. Cerchiamo in tabella nella colonna $L(t_i)$ il valore j per cui $L(t_j) \leq s_i \leq L(t_{j+1})$
3. Calcoliamo il rapporto che c'è fra la distanza da s_i a $L(t_j)$ e l'ampiezza totale dell'intervallo:
$$\alpha_i = \frac{s_i - L(t_j)}{L(t_{j+1}) - L(t_j)}$$
4. Calcoliamo $\hat{t_i}$, ovvero il parametro di valutazione della curva affinché la posizione su di essa risulti a distanza costante rispetto alla lunghezza della corda:
$$\hat{t_i} = t_j + \alpha_i(t_{j+1} - t_j)$$

Facciamo un esempio. Prendendo in riferimento la prima tabella (2.1), vogliamo calcolare la posizione sulla curva all'istante $i = 5$ (ovvero $\hat{t}_5$) se si assume velocità di percorrenza costante. La

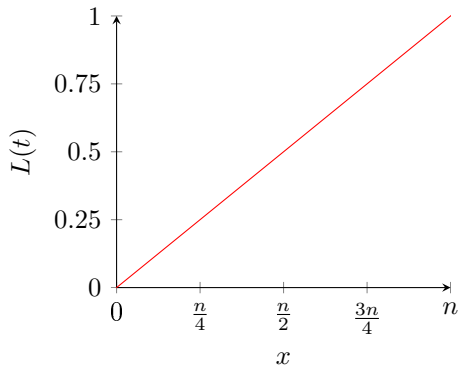
lunghezza percorsa (dipendente dal numero n) a tale istante è $s_5 = 0.01$ (tabella 2.2). Scorrendo la colonna $L(t_i)$ ci accorgiamo che s_5 è compreso tra L_{t_4} e L_{t_5}. Calcoliamo ora α_5, che è pari a $\alpha_5 = \frac{0.01-0.0095}{0.0108-0.0095} \approx 0.385$. Il nuovo punto di valutazione $\hat{t}_5$ sarà quindi in posizione $\hat{t}_5 = 0.0075 + 0.385(0.01 - 0.0075) = 0.00846$. La tabella 2.3 mostra una possibile configurazione di valori $\hat{t}_i$ a partire dalle tabelle precedenti.

Tabella 2.3: Look-up table 3

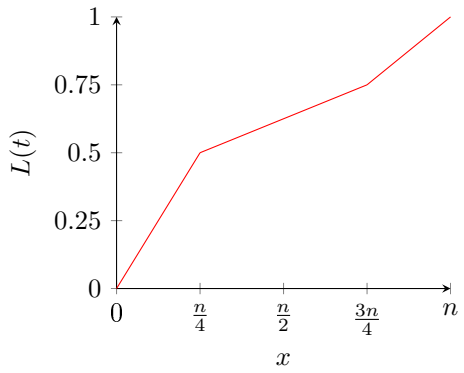
frame i	$\hat{t}_i$	$L(\hat{t}_i)$
1	0	0
2	0.00125	0.0025
3	0.0025	0.005
4	0.00451	0.0075
5	0.00846	0.01
⋮	⋮	⋮
⋮	⋮	⋮
398	0.9942	0.995
399	0.9979	0.9975
400	1	1

2.2.4 Controllo sulla velocità

La curva parametrizzata alla lunghezza d'arco ci permette di controllare facilmente la velocità di movimento di un corpo su di essa. La nostra animazione prevede n frame, tanti quanti i punti di valutazione del nostro percorso. Sappiamo inoltre che in esattamente n frame la camera passa dal punto iniziale al punto finale della traiettoria. Possiamo quindi affermare con sicurezza che la velocità media della telecamera sia circa pari a $\frac{1}{n}$ di curva percorsa in un frame.
Il software dispone di una finestra apposita per il controllo della velocità. Questa viene mostrata tramite grafico il cui asse delle ascisse rappresenta la successione dei frame e il valore di $L(t)$ ad ogni frame, normalizzata nell'intervallo $[0, 1]$.

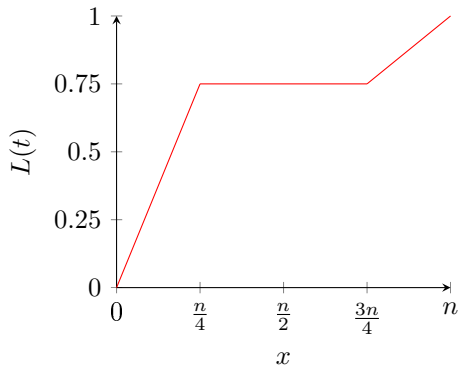


La funzione individuata è $f(x) = x$ e rappresenta un oggetto che si muove a velocità costante per tutta la curva. Infatti, in quanto la curva è parametrizzata alla lunghezza d'arco, ad ogni frame i si valuta precisamente $L(t_i)$. Il seguente grafico mostra invece una velocità che varia nel tempo:

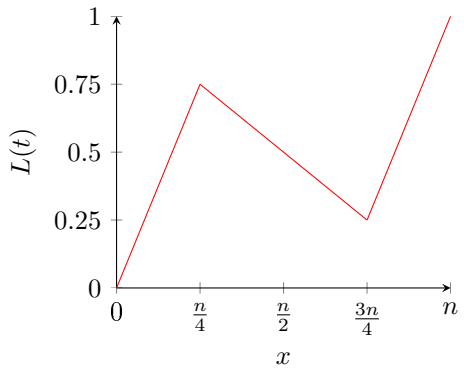


Esaminiamo il comportamento di tale funzione a tratti. Durante il primo quarto dell'animazione la camera è giunta a metà della curva ($f(x) = 2x$ in $[0, \frac{n}{4}]$). Questo significa che si è mossa al ritmo di due punti di valutazione ogni frame. In seguito ha però rallentato la sua corsa, impiegando ben due frame per un singolo punto t_i ($f(x) = \frac{x}{2}$ in $[\frac{n}{4}, \frac{3n}{4}]$). L'ultimo quarto è stato percorso a velocità costante.

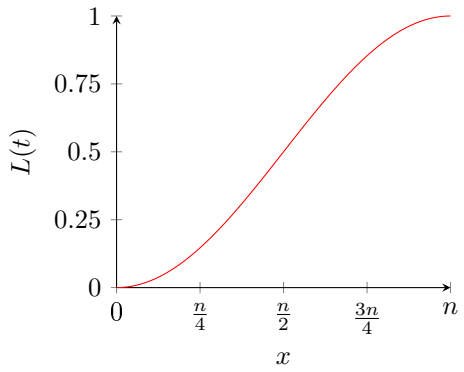
In altri casi invece siamo costretti a servirci della tabella contenenti i valori delle lunghezze ad arco. Questo accade quando tale funzione (o parti di essa) non ha andamento lineare. Analogamente a quanto fatto in precedenza, è sufficiente calcolare la percentuale di percorrenza al frame corrente per poi individuare, grazie alla tabella, il punto t_i per il quale $L(t_i)$ è la più vicina possibile con il valore che cerchiamo. L'eventuale parte in eccesso (o in effetto) verrà rimossa (o aggiunta) al prossimo frame. I prossimi esempi mostrano invece casi particolari. Il primo rappresenta un corpo che per alcuni frame rimane immobile:



Per ben metà del tempo la nostra telecamera apparirà ferma a esattamente $\frac{3}{4}$ della lunghezza totale della curva. Potenzialmente si può quindi arrivare al termine del percorso prima dell'ultimo frame, ma necessariamente la camera dovrà risultare ferma per un certo periodo di tempo.
È inoltre possibile fare in modo che la telecamera, arrivato ad un certo punto, torni indietro percorrendo la curva al contrario. Il nostro grafico mostrerà una funzione che ad un certo punto risulterà decrescente. Di seguito un basilare esempio:



Possiamo generalizzare tale metodo anche a funzioni che non hanno un grafico formato da spezzate. È sufficiente sostituire alla funzione lineare a tratti una più liscia di ordine superiore. Il software progettato per la tesi si serve proprio di una spline cubica per permettere all'utente di avere maggior controllo sulle variazioni di velocità. Il prossimo esempio rappresenta un andamento che ci obbliga ad eseguire l'algoritmo sopra descritto:



Nell'esempio mostrato in figura la curva *ease-in/ease-out*[1] ha la particolarità di iniziare con velocità bassa per poi accelerare per la maggior parte del tempo e infine rallentare dolcemente.

2.3 Orientamento

In questa sezione passerò in rassegna i metodi più interessanti per governare l'orientamento della telecamera. Durante la camera motion essa può rivolgere l'attenzione a tanti obiettivi diversi nella scena, sia che questi siano immobili oppure in movimento. Verranno quindi analizzate le tecniche più comuni per realizzare questi effetti.

2.3.1 Premessa sui tipi di obiettivi

In letteratura è facile trovare una sorta di classificazione delle animazioni di camera motion in base alla posizione dell'obiettivo inquadrato. Il lavoro di tesi da me svolto considera le seguenti tre:

- *Center of Interest (CoI)*
- *Look Ahead*
- *Follow Object*

Le animazioni Center of Interest (CoI) prevedono che gli obiettivi presenti sulla scena siano immobili nel tempo. La telecamera inquadra solitamente il baricentro di tali oggetti. Il modo per orientarsi da un oggetto all'altro durante l'animazione è spiegato più nel dettaglio nelle sottosezioni successive. Look Ahead prevede invece che l'obiettivo sia un oggetto in movimento posto davanti alla telecamera sulla stessa traiettoria. L'oggetto in questione si muove solitamente alla stessa velocità della camera in modo da mantenere la distanza decisa inizialmente.
Un'animazione di tipo Follow Object prevede un obiettivo che si muove su una curva diversa da

[1] Rick Parent. *Computer Animation: Algorithms and Techniques*. 2002.

quella della telecamera. La velocità di tale oggetto non deve essere necessariamente uguale a quella della camera.

2.3.2 Matrici di rotazione

La rotazione è una trasformazione affine che, fissato un punto di riferimento C detto *pivot* e un verso di rotazione, muove ogni punto di un oggetto nel verso assegnato in maniera tale che per ogni suo punto la distanza da C si conservi. Una rotazione di un angolo θ attorno all'origine degli assi O di un punto $P = (x, y) \in R^2$ genera un nuovo punto $P' = (x', y') \in R^2$ di coordinate:

$$x' = x\cos(\theta) - y\sin(\theta) \qquad y' = x\sin(\theta) + y\cos(\theta)$$

Il verso di rotazione è considerato antiorario se l'angolo è positivo, viceversa se negativo. In forma matriciale, vale la seguente relazione:

$$P' = RP \qquad P = \begin{bmatrix} x \\ y \end{bmatrix} \qquad P' = \begin{bmatrix} x' \\ y' \end{bmatrix}$$

dove la matrice di rotazione R è così definita:

$$R = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$$

La stessa trasformazione, in coordinate omogenee, è così espressa:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

La rotazione in R^3 è invece particolarmente complessa. È semplice ricavare le matrici di rotazione attorno ad uno dei tre assi principali ma una rotazione attorno ad un generico asse non consiste semplicemente nell'applicazione consecutiva delle tre suddette. Si deve ad Eulero la formulazione di un teorema che assicura che una qualsiasi rotazione in R^3 si può ottenere come composizione di esattamente 3 rotazioni attorno agli assi coordinati. Non scenderò nella dimostrazione di tale teorema e mi limiterò a fornire un'utilissima formula.
Definiamo le matrici base di rotazione attorno agli assi principali rispettivamente di un angolo α,

β e γ in questo modo:

$$R_x(\alpha) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\alpha) & -\sin(\alpha) & 0 \\ 0 & \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \; R_y(\beta) = \begin{bmatrix} \cos(\beta) & 0 & \sin(\beta) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\beta) & 0 & \cos(\beta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \; R_z(\gamma) = \begin{bmatrix} \cos(\gamma) & -\sin(\gamma) & 0 & 0 \\ \sin(\gamma) & \cos(\gamma) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Sia inoltre u il vettore attorno a cui ruota P e $v = \frac{u}{\|u\|}$ il suo versore.
La strategia da seguire consiste nell'effettuare due rotazioni per allineare v con l'asse z, ovvero $R_x(\alpha)$ e $R_y(\beta)$. A questo punto possiamo ruotare di γ attorno all'asse z tramite $R_z(\gamma)$ e per finire annullare le rotazioni per l'allineamento applicando $R_x(-\alpha)$ e $R_y(-\beta)$. La composizione finale delle matrici sarà pertanto la seguente:

$$R = R_x(-\alpha)R_y(-\beta)R_z(\gamma)R_y(\beta)R_x(\alpha)$$

Se l'oggetto non si trovava all'origine è necessario applicare prima una traslazione per portarlo in O, ruotarlo e infine riportarlo nella posizione di partenza.

2.3.3 Quaternioni

I quaternioni rappresentando un metodo avanzato per descrivere e manipolare le rotazioni servendosi dei numeri complessi. Sia i il numero immaginario pure tale che $i^2 = -1$. La famosa identità di Eulero

$$e^{i\theta} = \cos(\theta) + i\sin(\theta)$$

permette di scrivere un numero complesso in coordinate polari come

$$c = a + ib = re^{i\theta} \qquad \text{dove} \quad r = \sqrt{a^2 + b^2} \quad \text{e} \quad \theta = \arctan(\frac{b}{a})$$

a viene solitamente detta *parte reale* mentre b prende il nome di *parte immaginaria*. Supponiamo di voler ruotare c di un angolo ϕ intorno all'origine degli assi O. Il nuovo punto c' può essere espresso in coordinate polari nel seguente modo:

$$c' = re^{i(\theta+\phi)} = re^{i\theta}e^{i\phi}$$

Per una rotazione in due dimensioni è quindi sufficiente moltiplicare il punto originale per l'operatore di rotazione $e^{i\phi}$.
La rotazione in R^3 attorno ad O è invece più complicata, in quanto dobbiamo specificare sia una

direzione (un vettore) che la quantità di rotazione (uno scalare). La coppia ordinata formata dalla parte scalare e dal vettore è conosciuta con il nome di *quaternione*, dato dal matematico irlandese W. R. Hamilton che per primo li teorizzò nel 1843. Siano s la parte scalare e $v = (x, y, z)$ il vettore direzione. Esprimiamo il quaternione q che li rappresenta in questo modo:

$$q = (s; v)$$

Può essere altrimenti visto come una generalizzazione dei numeri complessi, dove s è la parte reale e le componenti di v sono tre parti complesse.

$$q = s + ix + jy + kz$$

dove i, j e k sono le parti immaginarie che soddisfanno le seguenti proprietà:

$$i^2 + j^2 + k^2 = -1$$
$$ij = -ji = k$$
$$jk = -kj = i$$
$$ki = -ik = j$$

Le operazioni sui quaternioni definiti in modo naturale sono:

- Prodotto di quaternione per uno scalare
- Somma fra due quaternioni
- Prodotto fra due quaternioni

Il prodotto fra due quaternioni q_1 e q_2 è dato dalla seguente formula:

$$q_1 q_2 = (s_1 s_2 - v_1 \bullet v_2; s_1 v_2 + s_2 v_1 + v_1 \times v_2)$$

Il prodotto è associativo ma non commutativo. Il modulo di un quaternione è pari a

$$\|q\| = s^2 + v \bullet v$$

mentre il suo inverso è

$$q^{-1} = \frac{1}{\|q\|^2}(s; -v)$$

Un quaternione di modulo pari ad 1 è detto *unitario*.
Ora che abbiamo introdotto la matematica alla base del concetto del quaternione vediamo come

farne uso per applicare una rotazione ad un punto. Data una rotazione di un angolo θ intorno ad un asse passante per l'origine O identificato dal versore u, definiamo il seguente *quaternione unitario* e il suo inverso:

$$q = (s; v) = (\cos(\frac{\theta}{2}), u\sin(\frac{\theta}{2})) \qquad q^{-1} = (\cos(\frac{\theta}{2}), -u\sin(\frac{\theta}{2}))$$

Dato un punto P di coordinate affini $(p; 1)$, il quaternione che lo rappresenta è questo:

$$r = (0; p)$$

L'operatore rotazione $R_q(r)$ sarà così definito:

$$R_q(r) = qrq^{-1}$$

È dimostrabile che la parte reale di $R_q(r)$ è nulla mentre la parte vettoriale è data da

$$p' = s^2 + v(p \bullet v) + 2s(v \times p) + v \times (v \times p)$$

devo s e v sono gli stessi del quaternione unitario. Il nuovo punto P' avrà coordinate pari a $(p'; 1)$, esattamente quello che avremmo ottenuto ruotando P di un angolo θ intorno ad u.

2.3.4 Frenet frame

Particolarmente interessenta è l'utilizzo dei cosiddetti *Frenet frame*. Il termine *frame* ha qui un significato completamente diverso da quello usato per descrivere le parti di un animazione. Non sarà però complicato capire la differenza tra i due contesti. R. Parent[2] ne dà una descrizione facilmente comprensibile.
Un Frenet frame è un sistema di coordinate (u, v, w) locale ad un singolo punto di una curva $C(t)$. Definito un qualsiasi punto $P = C(\bar{t})$ su di essa, il Frenet frame costruito su di esso dipende dalla tangente e dalla curvatura della traiettoria in quel punto. I tre vettori che lo definiscono sono ortogonali fra loro e normalizzati ed hanno le seguenti caratteristiche:

- w è rivolto nella direzione della derivata prima $C'(\bar{t})$.
- v è ortogonale a w nella direzione della derivata seconda $C''(\bar{t})$.
- u si ricava dal prodotto vettoriale tra w e v.

[2]Rick Parent. *Computer Animation: Algorithms and Techniques.* 2002.

Pertanto le formule per ricavare tali vettori sono le seguenti:

$$w = C'(\bar{t})$$
$$u = C'(\bar{t}) \times C''(\bar{t})$$
$$v = w \times u$$

I vettori u e v sono detti rispettivamente *normale* e *binormale*.

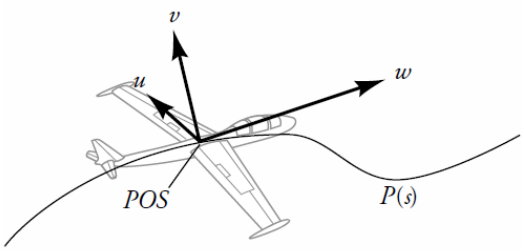


Figura 2.2: Frenet frame locale

Queste definizioni non tengono però in considerazione un problema che si potrebbe verificare: la curva potrebbe avere in alcuni punti curvatura nulla ($C''(t) = 0$). Questo significa che la normale v può essere indefinita. Quando ciò succede si è soliti interpolare il primo frame valido fra quelli precedenti e il primo valido tra quelli successivi. Parent ci mostra che non essendoci curvatura l'interpolazione fra questi due frame consiste in una semplice rotazione attorno w, il cui angolo è pari a $\theta = \arccos(v_1 \bullet v_2)$. Un altro problema simile si verifica quando ci sono discontinuità. In questi casi la derivata seconda cambia bruscamente segno e il nostro oggetto in movimento apparirà rivolto verso il basso. In questo caso si più utilizzare un vettore fisso che possa approssimare il valore della derivata seconda. La figura 2.3 mostra due possibili occasioni in cui si verificano i suddetti problemi.

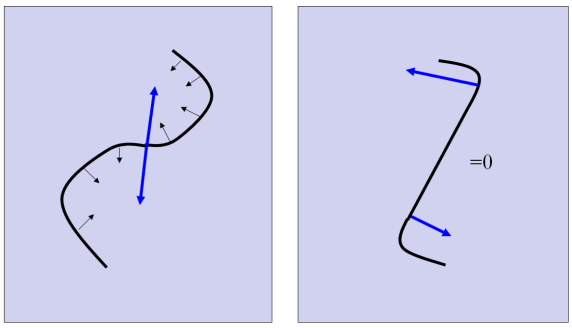


Figura 2.3: Problemi comuni nell'utilizzo dei Frenet frame quando la curvatura cambia segno in due punti successivi della curva (a sinistra) e quando questa è indefinita per un lungo intervallo (a destra)

Il software presentato si serve dei Frenet frame per l'orientamento del followed object sulla traiettoria. Tale oggetto è sempre orientato lungo la direzione w. La rotazione che vogliamo applicare può essere definita in forma matriciale in maniera non univoca. La matrice di riferimento utilizzata all'interno del software utilizzata è la seguente:

$$R = \begin{bmatrix} w_1 & w_2 & w_3 & 0 \\ u_1 & u_2 & u_3 & 0 \\ v_1 & v_2 & v_3 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Si è inoltre fatto in modo che la derivata seconda fosse in ogni punto pari al vettore $j = (0, 0, -1)$ per il motivo descritto in precedenza. Se si preferisce usare un vettore con componenti positive, sarà sufficiente definire v pari a $v = u \times w$. La scelta di diversi vettori produce diverse matrici, in quanto la direzione di normale e binormale cambierà di conseguenza.
I casi di seguito descritti sono in alternativa al Frenet frame e si possono applicare per orientare la telecamera verso un obiettivo posto in varie posizioni.

Center of Interest (CoI)

Supponiamo ora di voler orientarci verso un oggetto posto in una qualsiasi posizione. Il punto P' in questione sarà il nostro Center of Interest. Il vettore direzione sarà dato dalla differenza tra P' e P. Si è soliti inoltre servirsi di un *up-vector*, ovvero di un vettore che possa forzare la binormale v ad essere sempre rivolta verso l'alto. Seguendo l'esempio fatto da R. Parent e supponendo che tale direzione coincida con il vettore $up = (0, 1, 0)$, il frame sul punto P sarà il seguente:

$$w = P' - P$$
$$u = w \times up$$
$$v = u \times w$$

Look Ahead

Se il nostro obiettivo si trova sulla nostra stessa curva, P' sarà distante una certa lunghezza all'arco da P (figura 2.4). Possiamo quindi definire $P' = C(t+x)$, dove x rappresenta la distanza in termini di punti di valutazione. Le componenti del frame pertanto diventano:

$$w = P' - P = C(t + x) - P$$
$$u = w \times up$$
$$v = u \times w$$

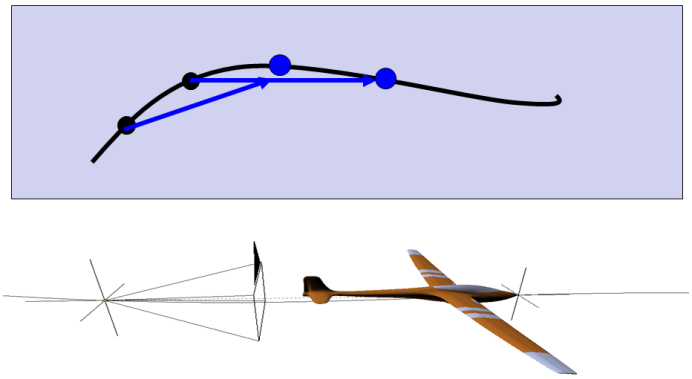


Figura 2.4: Look Ahead: l'oggetto obiettivo si trova sulla stessa curva della telecamera

Follow Object

Se invece il nostro obiettivo si sta spostando su una traiettoria differente da quello della telecamera è necessario considerare un up-vector generale. Sia $P(t)$ la curva sulla quale si muove la telecamera e $C(t)$ la seconda curva (figura 2.5). Possiamo inoltre definire una curva $U(t)$ i cui punti esprimono le direzioni dell'up-vector al tempo t. La nuova direzione generale dell'up-vector è data da $U(t) - P(t)$. Ne consegue quindi un frame calcolato in questa maniera:

$$w = C(t) - P(t)$$
$$u = w \times (U(t) - P(t))$$
$$v = u \times w$$

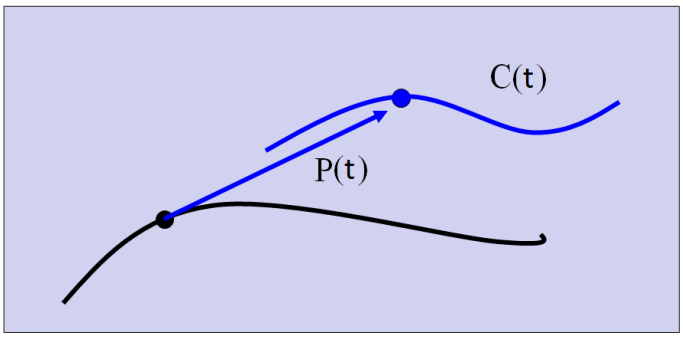


Figura 2.5: Follow Object: la telecamera in posizione $P(t)$ è orientata verso l'oggetto obiettivo posto su $C(t)$

2.3.5 Interpolazione lineare fra keyframe

Il software da me realizzato permette all'utente di avere controllo sull'orientamento della telecamera durante il suo movimento lungo il percorso. Questo consiste nella possibilità di specificare uno o più obiettivi che devono essere inquadrati dalla camera in un certo frame dell'animazione. Il CoI può quindi non essere unico. Definisco questi frame significativi con il nome di *keyframe*.
Questo termine è spesso usato in letteratura in maniera generale, perché i keyframe si prestano a molteplici utilizzi. Possiamo considerarli come delle strutture dati che ci permettono di legare ad un punto della curva una precisa condizione che deve risultare soddisfatta quando la telecamera si trova in quel punto. Le condizioni più frequenti che si scelgono - e di cui faccio uso in CaMoT - riguardano la velocità e/o l'orientamento.
Sia quindi $P = C(t)$ un punto della curva al tempo t. Rappresento il keyframe che lega posizione della telecamera, velocità ed orientamento in questa maniera:

$$K = \{P, v, F\}$$

dove v è il vettore velocità e F il frame che definisce un orientamento. Al punto P sulla curva la telecamera dovrà avere necessariamente velocità pari a v e un orientamento uguale a F.
All'interno del software tratto tuttavia le componenti della velocità e dell'orientamento in maniera separata.
La prima viene infatti gestita tramite appositi keyframe - come descritto al punto 2.2.4 - che non tengono in considerazione la direzione verso la quale la telecamera si deve rivolgere. Viene pertanto utilizzata una definizione più compatta di keyframe quale $K = \{P, v\}$. Data la funzione della velocità, possiamo considerare come keyframe i punti su di essa che controllano attivamente le sue caratteristiche principali, quali la continuità oppure l'inclinazione. Nel capitolo 4 ne do un esempio pratico quando mi soffermo sulla descrizione della finestra che controlla la velocità della camera lungo il percorso.
Analogamente, l'orientamento viene gestito senza considerare la velocità della telecamera ma solo in funzione dei frame che definiscono le diverse direzioni di orientamento. Il keyframe rappresenta in questo caso l'obiettivo della telecamera quando essa si trova su un punto della curva P e pertanto lo possiamo definire come $K = \{P, F\}$. Questa sottosezione si concentrerà su questo tipo di struttura.
I keyframe ci permettono di stabilire a priori diversi tipi di obiettivi che devono essere inquadrati dalla nostra telecamera in un certo istante di tempo. Non sappiamo però nulla sui valori intermedi tra due keyframe. L'obiettivo della nostra camera dovrà spostarsi nella maniera più fluida possibile dal primo al secondo. L'*interpolazione fra due keyframe* (o *tweening*) è l'operazione che cerchiamo. Dati due keyframe $K_1 = \{P_1, F_1\}$ e $K_2 = \{P_2, F_2\}$, l'interpolazione fra di essi consiste nello stabilire

una funzione per cui la camera, inizialmente in posizione P_1 e con un orientamento definito dalle coordinate di un frame F_1, muti l'inquadratura lungo il percorso in maniera tale da avere l'orientamento definito da F_2 nel momento esatto in cui essa giunga nel punto P_2. Una volta stabilita suddetta funzione, possiamo andare a valutarla nei punti intermedi tra i due keyframe e trovare di conseguenza il frame dell'orientamento della camera in ogni punto P compreso tra P_1 e P_2. Suddetta funzione può essere in linea teorica di qualsiasi tipo e avere le più svariate caratteristiche, a patto che non sia discontinua, che interpoli i due frame agli estremi e che mantenga ogni frame intermedio tra F_1 e F_2. Questo significa che possiamo esprimere tale funzione come una combinazione convessa tra i frame F_1 e F_2.
Il software fa uso di una combinazione lineare unita alla funzione OpenGL `gluLookAt`. Tale funzione permette di gestisce in maniera intuitiva la telecamera chiedendo allo sviluppatore di specificare semplicemente tre punti in input: la posizione, l'obiettivo da inquadrare e l'up-vector. In questa maniera, mantenendo l'up-vector costante, è sufficiente interpolare tra loro dei punti obiettivo $T \in R^3$. I suddetti keyframe possono pertanto essere ridefiniti come $K_1 = \{P_1, T_1\}$ e $K_2 = \{P_2, T_2\}$.
L'interpolazione lineare prevede che la funzione congiungente T_1 e T_2 sia equivalente ad un segmento. Considerando la curva parametrizzata all'arco, sia $P_1 = C(t_1)$ e $P_2 = C(t_2)$. Il punto T_t intermedio compreso fra T_1 e T_2 al tempo t viene calcolato come segue:

$$T_t = \alpha T_2 + (1 - \alpha) T_1$$

dove α è pari a:

$$\alpha = \frac{t - t_1}{t_2 - t_1} \qquad t_1 \leq t \leq t_2$$

La figura 2.6 mostra un esempio visivo del funzionamento dell'interpolazione lineare.
La scelta dell'interpolazione lineare è stata fatta essenzialmente per due motivi:

1. La velocità di interpolazione dipende solo dalla velocità con cui la camera percorre la curva.
2. È la più semplice funzione interpolante e ha costo computazionale minimo.

Dal punto di vista concreto, il software da me presentato valuta α ad ogni frame dell'animazione ed esegue l'interpolazione tra il keyframe precedente e quello successivo al punto di valutazione corrente. Stabilito il nuovo CoI, è necessario calcolare il nuovo Frenet frame locale alla camera. Il procedimento è lo stesso visto precedentemente e consiste nella determinazione dei tre assi w, u e v. Il metodo presentato nella sottosezione seguente è invece più generale e ha il vantaggio di non dovere necessariamente servirsi di un up-vector costante.

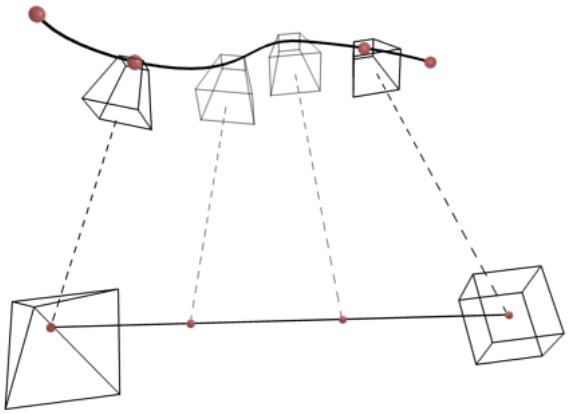


Figura 2.6: Interpolazione lineare tra keyframe

2.3.6 Interpolazione fra keyframe tramite quaternioni

Due keyframe possono essere interpolati anche servendosi dei quaternioni descritti precedentemente. Questo è attualmente il metodo più utilizzato in computer grafica, in quanto è semplice da realizzare e previene il problema del blocco cardanico (più frequentemente detto in inglese *gimbal lock*), che si verifica quando due assi di rotazione si allineano. Per evitare che il modulo dei quaternioni possa influire sull'interpolazione si è soliti utilizzare quaternioni unitari, che potremmo immaginare come dei punti su una sfera unitaria in quattro dimensioni.

Rappresentiamo due orientamenti tramite due quaternioni unitari q_1 e q_2 e cerchiamo di definire tramite interpolazione i quaternioni intermedi che permettono una transizione dell'orientamento da q_1 a q_2. In questo caso una interpolazione lineare non è la soluzione migliore e Parent[3] ci mostra il motivo. Tale interpolazione prevede che i quaternioni intermedi siano posti su un segmento che inizia in q_1 e termina in q_2. Calcolare questi quaternioni ad intervalli equidistanti sul segmento non produce una velocità di rotazione costante in quanto a suddetti intervalli ne sono associati altri sulla sfera unitaria che non risultano mai equidistanti. La figura 2.7 mostra una dimostrazione intuitiva di questo effetto su una sfera in due dimensioni.

[3]Rick Parent. *Computer Animation: Algorithms and Techniques*. 2002.

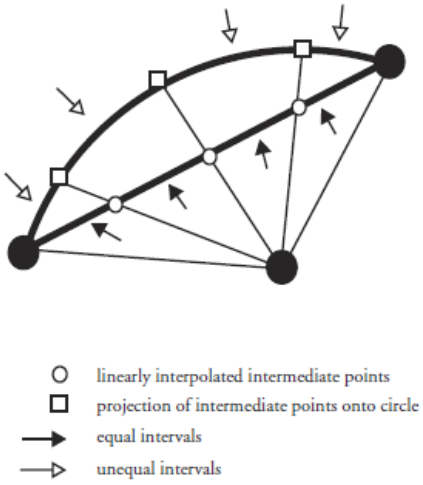


Figura 2.7: Dati due punti su una circonferenza, l'interpolazione lineare ad intervalli equidistanti su un segmento genera intervalli non equidistanti se proiettiamo i punti intermedi sulla circonferenza

Per avere velocità costante dobbiamo di conseguenza interpolare i due quaternioni direttamente sulla superficie della sfera unitaria. Il percorso più breve è l'arco che li congiunge definito sulla circonferenza con centro coincidente con il centro della sfera e passante per i due punti. Sappiamo tuttavia che un quaternione $q = (s; v)$ e il suo opposto $-q = (-s; -v)$ rappresentano lo stesso orientamento e pertanto dobbiamo stabilire se sia più breve il cammino che va da q_1 a q_2 oppure quello che va da q_1 a $-q_2$. Per farlo possiamo calcolare semplicemente il cosino dell'angolo θ compreso tra q_1 e q_2, in questo modo:

$$\cos\theta = q_1 \bullet q_2 = s_1 s_2 + v_1 \bullet v_2$$

Se il coseno è positivo, allora il percorso più breve è quello che separa q_1 da q_2. In caso contrario esso è quello compreso tra q_1 e $-q_2$.
Il nome di questo metodo di interpolazione è *slerp* (da *Spherical Linear intERPolation*) e venne trattato per la prima volta da K. Shoemake[4]. Dati due quaternioni che devono essere interpolati q_1 e q_2 e l'angolo θ tra loro compreso, qualsiasi quaternione intermedio q_t può essere definito con la seguente formula:

$$q_t = slerp(q_1, q_2, t) = \frac{\sin((1-t)\theta)q_1 + \sin(t\theta)q_2}{\sin\theta}$$

con il parametro $t \in [0, 1]$. Il quaternione risultante non è generalmente unitario. Ad ogni quaternione intermedio normalizzato corrisponde una matrice di rotazione che possiamo applicare alla telecamera. Sia $q_t = (s; v)$ un quaternione unitario dove s è la parte scalare e $v = (x, y, z)$ il vettore

[4]Ken Shoemake. *Animating Rotation with Quaternion Curves.* 1985.

direzione. La matrice di rotazione M_t corrispondente a q_t è la seguente:

$$M_t = \begin{bmatrix} 1-2y^2-2z^2 & 2xy-2zs & 2xz+2ys & 0 \\ 2xy+2zs & 1-2x^2-2z^2 & 2yz-2xs & 0 \\ 2xz-2ys & 2yz+2xs & 1-2x^2-2y^2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Capitolo 3

Modellazione e composizione della scena

3.1 Breve introduzione sulle mesh poligonali

Il lettore troverà spesso utilizzato nel corso di questo capitolo il termine *mesh*. In computer grafica si è soliti definire in tale modo un qualsiasi insieme di vertici, spigoli e facce che definisce la forma di un poliedro. Qualsiasi oggetto che si vuole rappresentare nella scena deve sempre possedere almeno le informazioni relative a tali componenti, in quanto necessarie per la rappresentazione della sua figura.

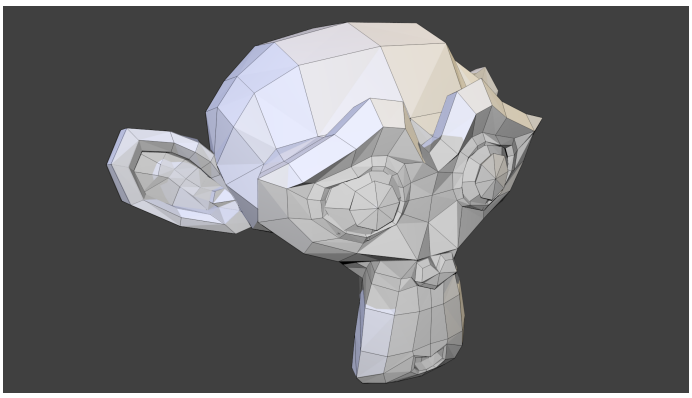


Figura 3.1: Suzanne, la mesh-mascotte di Blender

Un vertice (*vertex*, plurale *vertices*) è un punto in tre dimensioni che determina non solo la forma dell'oggetto ma anche caratteristiche utili aggiuntive per la corretta rappresentazione di colori, materiali o texture. Diversi modelli di illuminazione fanno uso delle normali calcolate sui vertici.

Uno spigolo (*edge*) è semplicemente un segmento che unisce due vertici.
Una faccia (*face*)è la superficie individuata da almeno tre spigoli. Queste possono essere un qualsiasi poligono ma si è soliti utilizzare - per questioni di performance e di stabilità geometrica e numerica - triangoli o, al limiti, quadrilateri. In questi due casi si parlerà, rispettivamente, di mesh triangolari o quadrangolari. Le facce sono le zone in cui si applicano le texture, solitamente tramite una parametrizzazione chiamata *UV mapping* o una sua evoluzione. Come per i vertici, anche le facce hanno una normale che permette di stabilirne l'orientamento e di calcolare l'intensità delle fonti luminose su di esse.
Definisco ora in maniera più formale il concetto di mesh triangolare. Sia M una varietà geometrica (*manifold*) a due dimensioni di un'arbitraria topologia inclusa in R^3. Una qualsiasi mesh di forma triangolare $s = (V, E, F)$ è una terna rappresentante una planarità a tratti approssimata di M. I valori della terna rappresentano rispettivamente i vertici, gli spigoli e le facce del poliedro tali che:

$$V = \{1, ..., N\}$$
$$E = \{(i, j) \in V \times V : X_j \in N(X_i)\}$$
$$F = \{(i, j, k) \in V \times V \times V : (i, j), (i, k), (k, j) \in E\}$$

Una mesh così definita descrive la topologia dell'oggetto che si vuole rappresentare ma non dà informazioni circa le sue proprietà geometriche. È pertanto necessario specificare le coordinate dei vertici all'interno dello spazio tridimensionale R^3.

3.2 Blender per la modellazione di mesh

Il motore grafico utilizzato per la realizzazione dei modelli (*mesh*) è Blender (versione 2.78). L'*engine* è completamente open source ed è continuamente aggiornato da parte della Blender Foundation. Due sono essenzialmente in motivi per il quale ho scelto di servirmi di Blender: il software è stato oggetto di diverse lezioni durante il corso di Computer Graphics - e questo mi ha permesso di acquisire una certa dimestichezza nel suo utilizzo - e permette, tra le tante cose, di poter esportare con facilità i modelli realizzati. Ritengo tuttavia necessario sottolineare che l'utilizzo di Blender sarebbe stato del tutto opzionale, in quanto OpenGL fornisce autonomamente tutte le primitive necessarie per poter creare qualsiasi tipo di forma. Definire vertice per vertice ogni modello sarebbe stato possibile. Avrei però pagato un costo in termini di tempo e di precisione.

Figura 3.2: Logo di Blender

Consideriamo ora un singolo modello e definiamone le caratteristiche di base. Per ragioni di performance ho optato per realizzare *mesh* composte da un numero di poligoni tali da offrire un buon compromesso tra la chiarezza della scena e la velocità di elaborazione della stessa da parte di OpenGL. Poiché la realizzazione di una scena visivamente appetibile non è l'obiettivo ultimo della mia tesi - ma ha solo lo scopo di mostrare all'utente le modifiche sull'animazione - i modelli realizzati non sono dotati di materiali, texture o di mappe UV. Essi possono essere percepiti come semplici strutture dati che indicano le coordinate dei vertici e quali di questi compongono le facce. Questo è tutto ciò che è sufficiente per poter esportare un poligono che possa essere disegnato.
Nella figura 3.3 si può vedere il modello di uno scrigno che fa parte della scena. Siamo in *Edit mode* e Blender ci permette di modificare ogni singolo vertice, spigolo o faccia a nostro piacimento. Vicino ad esso lo stesso oggetto, ma in modalità *Object mode*.

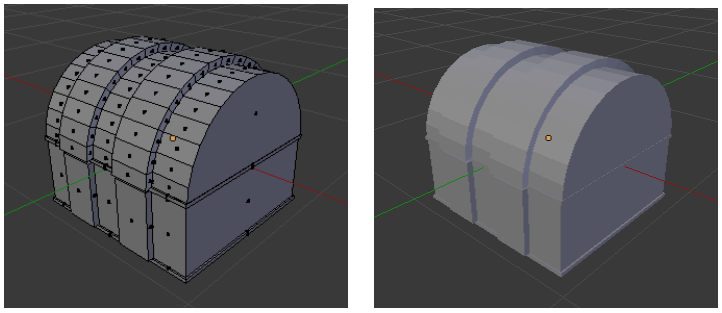


Figura 3.3: Scrigno visto in *Edit mode* e in *Object mode*

Le istruzioni per ricavare un file di testo con queste (e altre) informazioni sono semplici. Una volta realizzato il nostro modello è sufficiente esportarlo nel formato a noi più congeniale. Clicchiamo su *File* e successivamente passiamo il mouse sopra *Export...* per poi selezionare il tipo di file che desideriamo.
Ho scelto di servirmi di file con estensione *.obj*, ma non c'è un vero motivo dietro. In ogni caso il esso sarà di tipo testuale. Interessante è invece la grande quantità di elementi che è possibile ricavare da una singola mesh, come riportato nella figura 3.4.

3.3 Caricamento dei modelli

Il file .obj generato consiste in un elenco di vertici, di cui vengono indicate le componenti nello spazio, delle normali ai vertici e di facce, che indicano da quali vertici sono composte. Le righe sono contrassegnate da una stringa di testo che ci permette di distinguere i tre casi: v se si tratta di un vertice, vn se le componenti seguenti sono di una normale, f se si tratta di una faccia. A questo

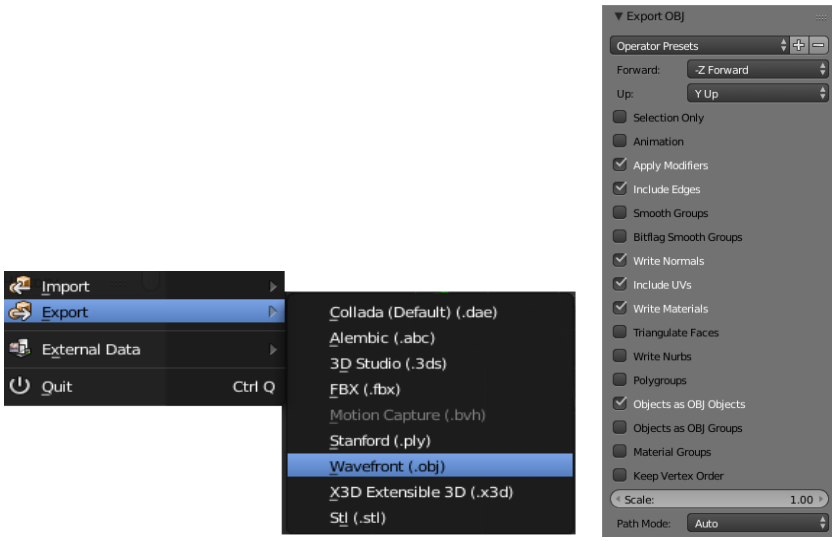


Figura 3.4: A sinistra, possiamo vedere le varie estensioni disponibili. A destra, possiamo scegliere quali componenti importare

punto sarà necessario scrivere un parser che, date quelle informazioni, crei un poligono che possa essere disegnato.
Una volta importato il file nel nostro progetto, Visual Studio è in grado autonomamente di mostrarci un'anteprima (figura 3.5). Questo è utile in fase di debug, perché ci permette di constatare rapidamente se l'oggetto è stato importato con successo.

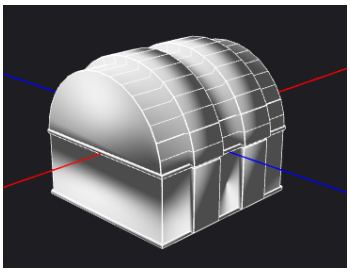


Figura 3.5: Scrigno mostrato da Visual Studio

A questo punto dobbiamo concretamente caricare in memoria la nostra mesh. Questa operazione viene eseguita solo una volta all'avvio del software e ogni poligono viene salvato in un'opportuna struttura dati. Di seguito viene mostrato il dettaglio implementativo, che si può trovare nel file `blenderObjectLoader.h`.
Una singola faccia è così definita:

```
typedef struct {
```

```
        std::vector<int> verticesInFace;
        std::vector<int> normalsInFace;
} POLYGON_FACE;
```

Questo è invece l'intero poligono:

```
typedef struct {
        int id;
        float cx;
        float cy;
        float cz;
        std::vector<POINT3D> vertices;
        std::vector<POLYGON_FACE> faces;
        std::vector<POINT3D> normals;
} BLENDER_OBJECT;
```

Notare che ogni poligono contiene variabili che lo identificano univocamente (`id`, utile per il *picking*) e che mantengono la propria posizione nella scena (`cx`, `cy` e `cz`). Queste ultime verranno definite in al momento del disegno.

Il file *.obj* viene letto fino in fondo, caricando nella struttura - riga per riga - vertici e facce (funzione `loadOBJ`). Quando la funzione `drawBlenderObject` viene chiamata il poligono viene disegnato, faccia dopo faccia. Lo pseudocodice seguente ha lo scopo di rendere più chiaro l'algoritmo e mostrare l'interazione con le primitive di OpenGL. Notare che viene mostrato solo il "cuore" della funzione: per uno studio più approfondito si consiglia di visualizzare direttamente il file `blenderObjectLoader.cpp`.

```
drawBlenderObject(BLENDER_OBJECT *obj) {
  int numFaces = obj->faces.size();
  for (int i = 0; i < numFaces; i++) {
    int numVertices = obj->faces.at(i).verticesInFace.size();
    glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
    glBegin(GL_POLYGON);
    for (int j = 0; j < numVertices; j++) {
      int v = ob->faces.at(i).verticesInFace.at(j);
      POINT3D vert = obj->vertices.at(v);
      int n = obj->faces.at(i).normalsInFace.at(j);
      POINT3D nor = obj->normals.at(n);
      glNormal3f(nor.x, nor.y, nor.z);
```

```
            glVertex3f(vert.x, vert.y, vert.z);
        }
        glEnd();
    }
}
```

3.4 Gestione delle collisioni

Un motore grafico che si rispetti dovrebbe permettere all'utente di avere massima libertà possibile nelle sue azioni e allo stesso cercare di metterlo in condizione di sfruttare al meglio tutte le funzionalità messe a disposizione. È pertanto per niente raro vedere consigli o suggerimenti quando si usa un software - professionale o non - dedicato alla modellazione o alle animazioni.
Una funzionalità che ritengo essere molto utile presente nel software presentato come tesi è la possibilità di notificare l'utente quando la curva da lui definita interseca le mesh della scena. Questo tipo di verifica è chiamata *collision detection*. Preciso subito che le collisioni di cui parlo sono quelle del tipo *curva-mesh*. Due mesh posizionate all'interno della scena possono collidere senza problemi. La letteratura offre in studio numerosi algoritmi ma ho deciso di servirmi di quello considerato essenzialmente il più semplice: l'uso delle *bounding box*.

3.4.1 Bounding box

Una tipica *bounding box* è un parallelepipedo rettangolo che contiene interamente una data mesh. Sono un caso particolare di *bounding volume*, macrocategoria che comprende qualsiasi forma in tre dimensioni che abbia lo scopo di delimitarne un'altra. Data un qualsiasi mesh siamo in grado di definire infiniti parallelepipedi che soddisfano questa condizione. Cerchiamo pertanto di dare qualche condizione aggiuntiva per perfezionare la nostra scatola. È bene notare che il nostro obiettivo dovrebbe essere ricercare quella forma che meglio approssima la mesh. Due condizioni che solitamente si ricercano sono le seguenti:

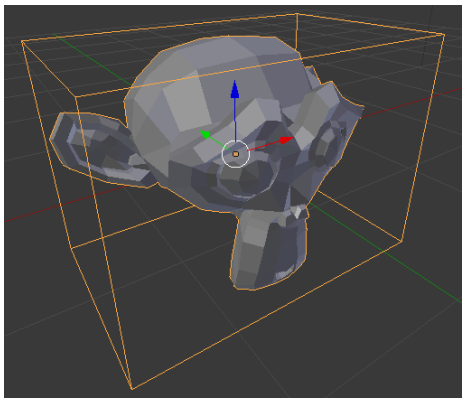


Figura 3.6: *Bounding box* in Blender

- La nostra scatola deve essere una *minimum bounding box*, ovvero il parallelepipedo rettangolo che ci interessa deve contenere al suo interno la totalità dei punti della mesh e, allo stesso tempo, occupare il volume minore possibile.

- Per ragioni di semplicità - anche a discapito di un possibile leggero calo delle performance - cerchiamo una *axis-aligned bounding box* (in gergo tecnico AABB), ovvero tale da avere gli assi allineati con quelli locali della mesh.

Queste due condizioni sono semplici da realizzare quando si ha a disposizione l'elenco dei vertici che definiscono la figura di partenza. Tuttavia, il risultato potrebbe essere particolarmente insoddisfacente e cercherò di spiegare brevemente il motivo.
Generalmente si considera tanto più buona la forma esterna quanto efficacemente essa riduca lo spazio vuoto che si forma tra di essa e quella interna. La superficie che non è occupata dalla figura inscritta produrrà irrimediabilmente una collisione "falsa", ovvero tale da essere rilevata dai nostri algoritmi ma non reale. Poiché è la forma della mesh a determinare quella del *bounding volume*, a questa deve essere dedicata grande attenzione. In ultima analisi, la figura inscritta è quella che determina la scelta del volume di collisione.
Abbiamo già specificato che per ragioni di performance ogni mesh è dotato di una *bounding box* dalla forma di un parallelepipedo rettangolo. Questa condizione è definitiva. Consideriamo per semplicità lo stesso problema ma in sole due dimensioni. La nostra scatola si trasforma in un rettangolo (*bounding rectangle*). Proviamo quindi ad immaginare come questa si possa comportare in relazione alla forma della figura interna. Se la mesh inscritta è un quadrato oppure un rettangolo, il volume di collisione può ricoprirla perfettamente e garantire che non ci siano superfici "vuote" al suo interno. Per ogni altra forma, al contrario, non si può dire la stessa cosa. Basti pensare ad un semplice triangolo: qualsiasi *bounding rectangle* produrrà tanto spazio vuoto quanto grande è la sua area. Il rapporto tra area occupata ed area non occupata è destinato a crescere per figure dotate di particolare convessità. Parimenti si può dire di forme in tre dimensioni e del loro volume. La soluzione più intuitiva a questo tipo di problema consiste nell'utilizzare più *bounding box* per una singola mesh la cui forma genera questa spiacevole situazione. La figura interna viene idealmente suddivisa in più parti tali da permetterci di inserire ognuna di esse all'interno di un parallelepipedo rettangolo, come mostrato nella figura 3.7. Non è più pertanto necessario che una singola *bounding box* comprenda dentro di sé tutti i punti della figura ma rimane vera la condizione per cui ogni punto, preso individualmente, faccia parte di almeno un volume di delimitazione.

3.4.2 Ottimizzazione tramite bounding volume hierarchy

Consideriamo ora un gruppo di mesh facenti parte della scena che vogliamo rappresentare. Ognuna di esse è dotata di bounding box e questo ci obbliga, generalmente, a considerarle tutte quando verifichiamo se la nostra curva può collidere con esse. Proviamo a ragionare sulla complessità com-

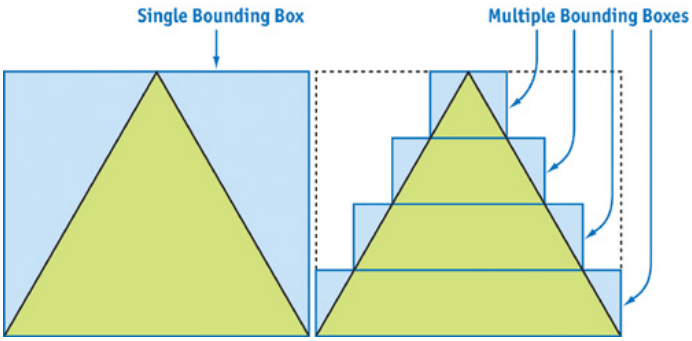


Figura 3.7: Utilizzo di *bounding box* multiple

putazione di questa operazione: per ogni singolo punto che definisce la nostra traiettoria dobbiamo verificare le condizioni di intersezione con ogni figura della scena. Questo costo - in termini di tempo - può essere particolarmente gravoso se il numero di punti o di mesh cresce.
La possibilità di poter raggruppare le mesh in gruppi in relazione alla distanza fra di esse facilità enormemente le cose. Questo è inoltre quasi sempre possibile se siamo noi stessi gli autori della scena e conosciamo a priori le posizioni di ognuna di esse. Si possono presentare essenzialmente due casi, che discuterò individualmente:

1. Tutte le mesh della scena sono sufficientemente vicine da poter essere raggruppate insieme.
2. Le mesh sono troppo lontane fra di loro per formare un unico grande gruppo

Nel primo caso la soluzione più efficiente consiste nella creazione di una nuova *minimum bounding box* che includa al suo interno tutte le mesh, come se queste fossero fuse in un unico oggetto. Apparentemente questa operazione sembra determinare un peggioramento delle prestazione in quanto aumenta il numero di bounding box da considerare. Tuttavia ora abbiamo un'informazione in più che risulterà importantissima: se un punto della curva non interseca la scatola contenitore allora certamente non potrà collidere con nessuna delle mesh al suo interno. Pertanto la prima collisione che si cerca di verificare è quella con la bounding box più grande e se questa risulta falsa allora non dobbiamo analizzare anche le altre. Per apprezzare maggiormente questo fatto proviamo ad immaginare una scena composta da circa 100 elementi sufficientemente vicini tra loro. Dato un qualsiasi punto della nostra traiettoria quante collisioni dobbiamo verificare? Consideriamo quella con la bounding box più esterna e solo in caso negativo tutte le altre. Nel caso peggiore saranno in totale 101 ma nel caso migliore si calcolerà una sola intersezione.
Il secondo caso merita invece un'attenzione particolare. Consideriamo ora tanti gruppi di mesh che sono particolarmente lontani fra loro. Creare una bounding box che comprenda le mesh di tutti non ci garantirà risultati tremendamente migliori, in quanto questa sarà di grandi dimensioni e pertanto la probabilità che ogni punto collida con essa sarà alta. Questo approccio in combinazione

con quello visto nel primo punto non è però da scartare: possiamo sempre definire dei sottoinsiemi di questa grande scatola. Questi sottoinsiemi potrebbero contenere una certa frazione dei gruppi di mesh. Possiamo continuare: all'interno di un sottoinsieme se ne vengono a formare altri più piccoli, ognuno dei quali bounding box di un numero minore di mesh. La struttura dati che si viene a formare è quella di un albero di bounding box che in letteratura prende il nome di *bounding volume hierarchy*.

Il funzionamento è analogo a quello dell'algoritmo descritto al punto uno, con la solo differenza che viene reiterato per ogni scatola all'interno di una bounding box di cui abbiamo verificato la effettiva collisione, seguendo una precisa gerarchia. Nel momento in cui la verifica di un'intersezione tra curva e contenitore produce un risultato negativo l'algoritmo si arresta e non prosegue per le bounding box interne. Le figure seguenti (3.8) mostrano una visione in due e tre dimensioni di una *bounding volume hierarchy*.

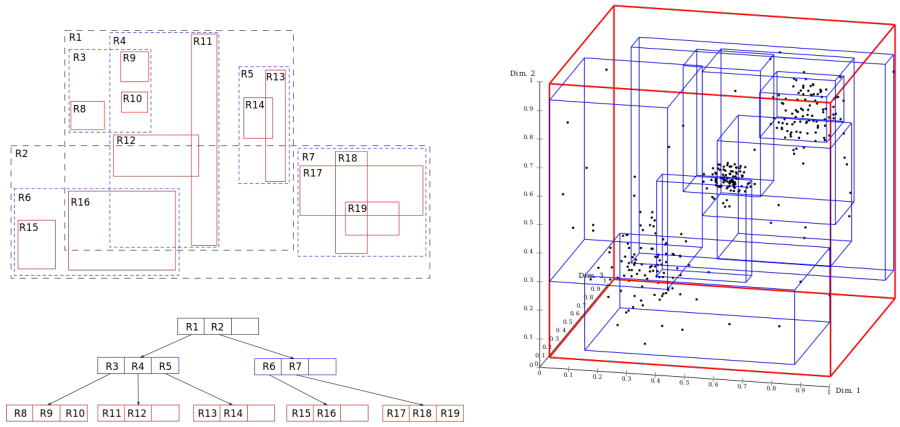


Figura 3.8: Visione bidimensionale (a sinistra) e tridimensionale (a destra) di una *bounding volume hierarchy*

3.5 Composizione della scena in OpenGL

Questa breve sezione riassume il modo in cui la scena è stata composta e l'ordine delle operazioni eseguita su ciascuna mesh. A queste si aggiunge la creazione delle *bounding box* e l'identificazione per il *picking* degli oggetti. Le modalità di funzionamento e le istruzioni dettagliate possono essere trovate nelle altre sezioni di questo capitolo. L'ordine è essenzialmente il seguente:

1. Le mesh vengono caricate individualmente in una struttura dati apposita che mantiene informazioni riguardo vertici e facce.

2. Vengono applicate trasformazioni elementari - traslazione, scalatura e rotazione - dove necessario per posizionare gli oggetti in scena.
3. Viene disegnata la mesh due volte: la prima volta viene colorato il poligono mentre la seconda viene tratteggiato il contorno.
4. Per ogni mesh si calcola la propria *bounding box*.
5. Se si ritiene che possa portare vantaggi, si crea un apposito *bounding volume hierarchy* con "contenitori" aggiuntivi.
6. Si definisce univocamente ogni oggetto tramite id, così che possa essere riconosciuto in fase di *picking*.

Le istruzioni ai punti 4 e 5 potrebbero - a discrezione dello sviluppatore - essere precedute senza sostanziali differenze dalle istruzioni al punto 6.
La figura 3.9 mostra la scena che si presenta all'utente all'avvio del software. Due semplici luci decorano la scena.

Figura 3.9: Scena 3D all'avvio del software

Sono state utilizzate diverse mesh, al fine di realizzare un ambiente nel quale potessero esserci tanti obiettivi da inquadrare. Queste comprendono:

- Il mare, sfondo della nostra scena.
- Un faro, composto da tanti piani e dotato di una porta. La parte superiore comprende un balcone, la lanterna e il tetto piramidale.

- Una casetta, dotata di tetto, porta e camino.
- Una scogliera, per metà sommersa dal mare.
- Un pontile, formato da tronchi di legno.
- Due boe, a forma di cartello (non visibili all'avvio del software).
- Una caverna, che riaffiora dal mare (non visibile all'avvio del software).
- Uno scrigno, nel cuore della caverna (non visibile all'avvio del software).
- Un aeroplanino di colore blu (il nostro obiettivo in modalità Follow Object e Look Ahead).

Capitolo 4

Progetto CaMot

4.1 Genesi del progetto e schema generale

Il progetto prende il nome di CaMot (sincrasi delle parole "camera" e "motion"). La genesi del software parte da molto lontano e attraversa varie fasi da sviluppo. Inizialmente concepito come esercizio/consegna riguardante le curve spline per l'esame di Computer Graphics, CaMot ha subito la prima grande trasformazione durante il tirocinio interno seguito dalle relatrici Serena Morigi e Damiana Lazzaro. Da semplice editor di curve, l'applicazione è diventata ben presto un tentativo di realizzare un basilare motore grafico che potesse permettermi di studiare a fondo il modo per creare animazioni di *camera motion*. Conclusasi tale esperienza dopo qualche mese di lavoro, decisi di continuare a mettere mano al codice al fine di aggiungere più funzionalità possibili. Nel frattempo cercai di apprendere nel dettaglio tutte quelle nozioni di tipo matematico che sarebbero state la base del software. Conoscere al meglio tali concetti è sempre stato il mio obiettivo sin dall'inizio. Posso ora, dopo centinaia di ore di lavoro, consapevole dei tanti possibili miglioramenti, definirmi soddisfatto del progetto da me realizzato. In questa sezione verrà analizzata principalmente l'interfaccia utente, con alcune sottosezioni dedicate agli strumenti che hanno reso possibile CaMot.
Viene di seguito anticipato lo schema del progetto, che mostra con una visione d'insieme tutto ciò che è stato utilizzato nella realizzazione del software.

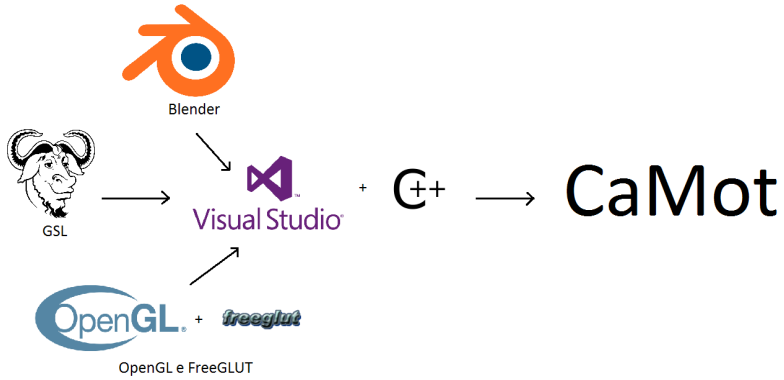


Figura 4.1: Schema riassuntivo del progetto CaMot

4.2 Tecnologie utilizzate

4.2.1 Ambiente di sviluppo

Il software è scritto interamente in C++ (richiesto dalle librerie OpenGL) ed è stato sviluppato per mezzo dell'ambiente di sviluppo Visual Studio Enterprise 2015. Le mesh - importate tramite semplici file testuali - sono state modellate tramite Blender (v. 2.78). Il sistema operativo sul quale è avvenuto lo sviluppo è Windows 8.1. Non è pertanto garantito il funzionamento su altri sistemi quali Linux (nelle varie distribuzioni) o MacOS.

4.2.2 Librerie utilizzate

OpenGL, GLUT e FreeGLUT

La libreria principale sul quale il software si basa è OpenGL (Open Graphics Library). Il termine "libreria" è usato in maniera impropria: una qualsiasi versione di OpenGL è piuttosto un'interfaccia software di programmazione (in gergo tecnico una vera e propria API) che specifica nel dettaglio un insieme variegato di funzionalità di tipo grafico che devono essere messe a disposizione dello sviluppatore. Ogni produttore di hardware dedicato (quali ad esempio le GPU) ha il compito di fornire l'implementazione di tali funzionalità. I dettagli sono però nascosti allo sviluppatore, che si serve di alcune primitive per eseguire qualsiasi operazione. La versione utilizzata dal software è la 4.2.0 ma - per ragioni di semplicità e per sfruttare ciò che ho appreso durante il corso di Computer Graphics - essa viene sfruttata in modalità compatibile alla versione 2.0. Un passaggio completo alla versione più avanzata o alla recente Vulkan (entrambe gestite dal Gruppo Khronos) potrebbe essere un obiettivo da perseguire nel futuro. OpenGL 2.0 si è rivelata particolarmente immediata nella

gestione della pipeline di rendering, nelle trasformazioni geometriche sulle mesh e nel controllo della telecamera. Questa è inoltre la prima che introduce un linguaggio specifico per la programmazione degli *shader* (GLSL).
Un programma scritto in OpenGL lavora tramite funzioni che lo sviluppatore decide di chiamare al momento opportuno e reagisce ad eventi generati dalle azioni dell'utente. La risposta in output è di tipo grafico: OpenGL ordina alla scheda grafica di risolvere determinate istruzioni in base alla propria implementazione e di scrivere sul *frame buffer* ciò che sarà mostrato sullo schermo. Le funzioni possono essere di vari tipi e ne elenco brevemente una classificazione delle più importanti:

- Funzioni primitive, che definiscono gli oggetti alla base di qualunque scena quali punti, segmenti, poligoni, curve, e tanti altri.
- Funzioni di trasformazione, che permettono traslazioni, scalature e rotazioni.
- Funzioni di visualizzazione, che descrivono in maniera rapida l'orientamento della telecamera e impostano le dimensione del *viewport* (la parte della scena visibile sullo schermo).
- Funzioni di input, che gestiscono l'interazione con l'utente, principalmente tramite mouse e tastiera.
- Funzioni di controllo, per permettere l'inizializzazione del programma, la gestione degli errori e delle varie finestre.

Il software da me realizzato fa uso di tre finestre (*window*) ognuna delle quali soddisfa un'esigenza diversa. Per la gestione di tali interfacce mi sono servito di una libreria aggiuntiva spesso usata con OpenGL chiamata GLUT (GL Utility Toolkit). Questa tratta ogni finestra come una struttura dotata di una coda di eventi che viene riempita quando l'utente compie una certa azione, quale ad esempio un click oppure un trascinamento o ancora la pressione di un tasto sulla tastiera e molte altre. Gli eventi vengono gestiti uno dopo l'altro secondo l'ordine di arrivo e attivano - se registrate - precise funzioni di callback che si occupano di attuare una risposta. Lo sviluppatore deve quindi implementare tali funzioni e pensare al proprio software come ad una macchina a stati che reagisce agli input dell'utente.
GLUT non è però aggiornata da almeno 20 anni e non è open source. Pertanto mi sono servito di una libreria sostitutiva molto più recente denominata FreeGLUT. L'intercambiabilità fra le due librerie è rafforzata dal fatto che FreeGLUT mantiene lo stesso nome per le stesse funzioni presenti in GLUT.

GNU Scientific Library (GSL)

Il programma fa uso della libreria open source GNU Scientific Library per l'approssimazione ai minimi quadrati tramite le funzioni B-spline. La versione attuale è al momento la numero 2.4. La

libreria permette un gran numero di funzionalità nelle varie aree della matematica. La documentazione asserisce che ci siano oltre un migliaio di funzioni rese facilmente disponibili allo sviluppatore. Queste sono divise per file in base al tipo di problema che aiutano a risolvere.
Il software si serve di GSL nel complesso di tre file: `gsl_blas.h` (per le operazioni tra matrici e vettori), `gsl_linalg.h` (per la decomposizione di Cholesky) e `gsl_poly.h` (per la valutazione della curva approssimante tramite matrice di Vandermonde).

4.3 Strutture dati ed elenco file

Questa sezione si concentra su alcune strutture dati di cui mi sono servito durante la programmazione del software. Verranno poi elencati i file che compongono materialmente il progetto.

Strutture dati

La struttura dati basilare è senza dubbio quella del punto in tre dimensioni. Esiste anche la versione 2D, utile quando non si vuole sprecare inutilmente spazio in memoria per una componente che non verrà utilizzata. Per ragioni di compatibilità con alcune librerie o per questioni di performance, gli array di punti sono quasi sempre stati spezzati in tre array, ognuno delle quali gestisce una singola componente.
Nel capitolo 3 ho parlato di due strutture dati. La prima rappresenta un oggetto importato da Blender, già ampiamente discusso. Il secondo è la *bounding box* che racchiude una mesh per la rilevazione delle collisioni.

```
typedef struct {
        float minX;
        float maxX;
        float minY;
        float maxY;
        float minZ;
        float maxZ;
} BOUNDINGBOX;
```

Tramite questi sei campi posso sempre ricavare la posizione degli otto vertici del parallelepipedo rettangolo allineato agli essi che delimita un oggetto.
Nel corso del capitolo 4 ho introdotto il concetto di keyframe per gestire l'orientamento della telecamera. La struttura dati relativa è la seguente:

```
typedef struct {
```

```
        POINT3D point;
        int position;
} KEYFRAME;
```

Il campo `point` rappresenta il punto della spazio 3D verso il quale la camera deve rivolgersi, mentre `position` rappresenta il punto sul percorso. La trasformazione graduale dell'orientamento tra due keyframe è stato descritto al punto 2.3.5 e si rimanda alla figura 4.5 per un esempio visivo.

Elenco dei file

Il software è composto da diversi file e di seguito ne faccio un rapido elenco classificandoli per estensione. I file `.cpp` sono i seguenti:

- `CaMot.cpp`, contenente la maggior parte del codice e il `main`.
- `parameterizations.cpp`, contenente le funzioni di parametrizzazione usate per le partizioni nodali e per la curva in generale.
- `boundingBox.cpp`, per l'utilizzo delle *bounding box*.
- `keyframe.cpp`, per l'interpolazione tra due keyframe.
- `blenderObjectLoader.cpp`, per il caricamento delle mesh realizzate in Blender.
- `matrixUtils.cpp`, funzioni utili dedicate alla matrici.
- `textUtils.cpp`, funzioni per la gestione del testo all'interno delle finestre.

Per ognuno di questi file - eccetto `CaMot.cpp` - esiste un relativo file `.h` contenente le strutture dati e le dichiarazioni dei metodi. Ulteriori file di questo tipo sono i seguenti:

- `constants.h`, che contiene le costanti per tutti le varie funzionalità del software.
- `structs.h`, contenente le strutture dati `POINT2D` e `POINT3D`.
- `strings.h`, per la definizione delle parti testuali.
- `colors.h`, per la gestione dei colori delle mesh.

Tutte le mesh utilizzate, di cui ometto l'elenco, hanno estensione `.obj`.

4.4 Intefaccia utente

4.4.1 Spline Editor Window

Questa finestra permette all'utente di creare a suo piacimento il percorso che la telecamera dovrà seguire durante l'animazione. La traiettoria è definita in base ai punti controllo che si decide di

inserire. I punti vengono aggiunti tramite mouse (tasto sinistro) nella posizione desiderata. Poiché risulta particolarmente difficile inserire dei punti in uno spazio tridimensionale ho deciso di servirmi di una finestra in sole due dimensioni e dare successivamente la possibilità all'utente di modificare la componente Y rimanente. All'avvio del software la finestra mostra gli assi X (in rosso) e Z (in blu) e la loro origine (in giallo). La componente Y di default assegnata all'inserimento del punto è pari a 220. La posizione della telecamera sarà sempre indicata con il colore grigio. In basso a destra è possibile vedere la lista di alcuni comandi, che verrà esaminata nel dettaglio successivamente.

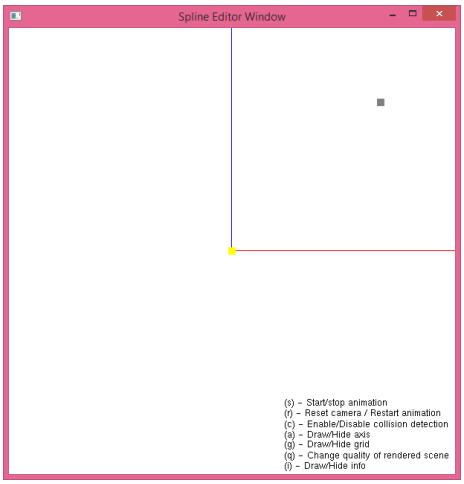


Figura 4.2: Spline Editor Window all'avvio del software

La gestione di tutto ciò che concerne la forma del percorso e il tipo di obiettivo della camera durante l'animazione viene controllata tramite menù, che l'utente può esaminare tramite click con il tasto destro del mouse. La figura 4.3 mostra l'apertura del menù principale. Ogni voce rappresenta un sottomenù.

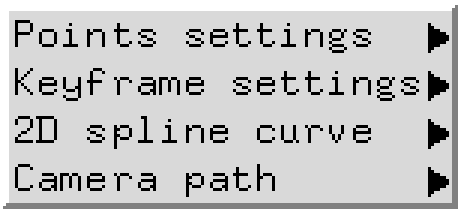
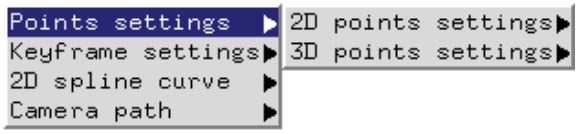


Figura 4.3: Menù pricipale

I quattro sottomenù sono *Points settings*, *Keyframe settings*, *2D spline curve* e infine *Camera path*. Esaminiamoli nello specifico.

Points setting

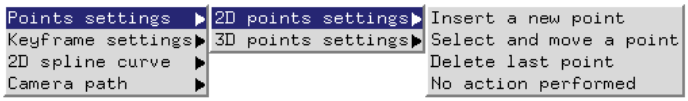
Il sottomenù *Points setting* permette all'utente di inserire, modificare, spostare od eliminare i punti da approssimare. Questi punti vengono utilizzati sia per definire le spline (diventando parte del poligono di controllo) che per realizzare le curve approssimanti nel senso dei minimi quadrati.



Points settings è a sua volta composto da due sottomenù: *2D points settings* e *3D points settings*.

2D points settings

Questo sottomenù controlla le modalità di inserimento, eliminazione e spostamento dei punti inseriti dall'utente.



Tramite l'opzione *Insert a new point* il software entra in modalità inserimento. L'utente può quindi aggiungere i punti (in verde). Non appena saranno presenti almeno 4 punti la curva approssimante verrà disegnata. Una linea tratteggiata mostra invece il poligono di controllo (che scompare se viene richiesta una curva approssimante nel senso dei minimi quadrati).

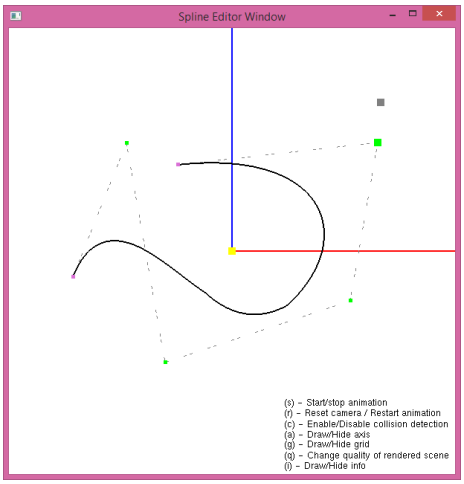
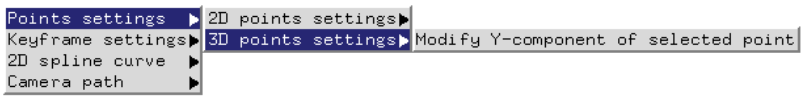


Figura 4.4: Inserimento di alcuni punti di controllo e curva approssimante

La modalità di spostamento, che si attiva tramite l'opzione *Select and move a point*, permette la modifica della posizione di un punto. Per selezionare il punto desiderato è sufficiente un click nelle sue immediate vicinanze. L'aumento delle dimensioni di tale punto certifica l'avvenuta selezione. A questo punto l'utente può trascinarlo fino alla posizione desiderata.
L'opzione *Delete last point* fa passare il software alla modalità di eliminazione. Ad ogni click sulla finestra corrisponde l'eliminazione dell'ultimo punto inserito.
Infine l'opzione di utilità *No action performed* disabilità la modalità corrente. Cliccare con il tasto sinistro all'interno della finestra non avrà alcun effetto. Utile se non si vuole rischiare di inserire, spostare o eliminare inavvertitamente dei punti.

3D points settings

Il sottomenù *3D points settings* consta di un'unica opzione disponibile.



L'opzione *Modify Y-component of selected point* permette all'utente di modificare la componente Y del punto di controllo attualmente selezionato. Ricordiamo che al momento dell'inserimento tale componente è pari a 220 e quindi la curva è parallela al piano XZ. Il valore attuale è mostrato in verde sopra ogni punto di controllo e può essere cambiato tramite la pressione dei tasti [+] e [-] del tastierino numerico.

Keyframe settings

Il sottomenù *Keyframe settings* gestisce invece lo spostamento e l'eliminazione dei keyframe inseriti tramite la Scene Window.

I keyframe appaiono come punti di colore viola. Al momento della creazione della curva saranno presenti due keyframe agli estremi, indicanti l'orientamento della telecamera quando questa si trova all'inizio e alla fine del percorso. Questi keyframe non potranno essere né selezionati né eliminati. Sarà però possibile cambiare l'orientamento in quel punto tramite la finestra della scena.
All'inserimento di un keyframe questo apparirà esattamente al centro della curva. L'opzione *Select and move a point* permette di selezionare un keyframe (che aumenterà visivamente di dimensioni)

e di spostarlo lungo la curva. I tasti [+] e [-] del tastierino numerico muovono rispettivamente in avanti e all'indietro il keyframe selezionato. Questi non potranno tuttavia superare il keyframe successivo né il precedente.

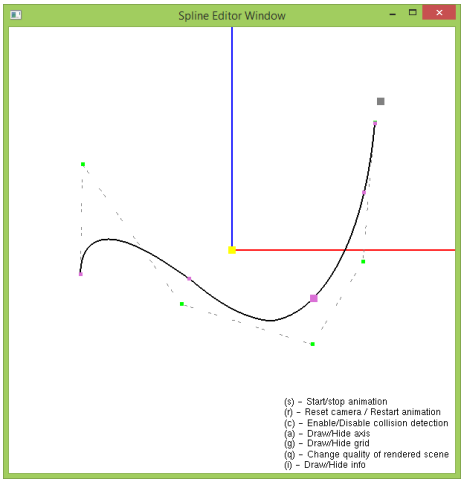


Figura 4.5: Curva con 3 keyframe non estremi, di cui il secondo attualmente selezionato

Le opzioni *Delete last point* e *No action performed* sono analoghe a quelle viste per i punti controllo.

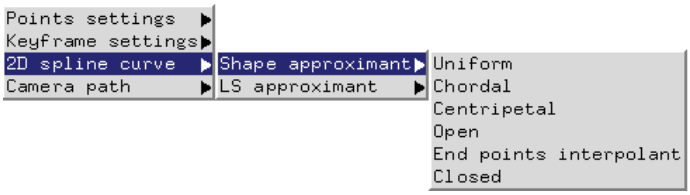
2D spline curve

Posizionando il cursore sopra questa voce si apre un ulteriore sottomenù tramite il quale l'utente può gestire le caratteristiche del percorso della telecamera.

È diviso a sua volta in *Shape approximant* e *LS approximant*.

Shape approximant

Questo sottomenù permette di variare la posizione dei nodi veri e di quelli fittizi nella partizione nodale estesa causando una modifica nella forma della curva. Per una migliore comprensione circa le possibili scelte e le loro conseguenza si consiglia di leggere al punto 1.1.8.



L'opzione di default è *Uniform*. Tale parametrizzazione non tiene conto della distanza tra i punti di controllo. Tali distanze sono invece considerate nella parametrizzazione della corda (*Chordal*) e in quella centripeta (*Centripetal*). La figura 4.6 mostra una curva definita dagli stessi punti di controllo ma parametrizzata in modi diversi.

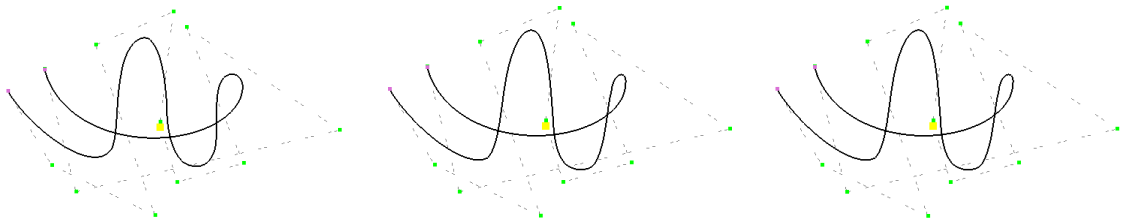


Figura 4.6: Parametrizzazione uniforme, della corda e centripeta dato lo stesso poligono di controllo

Al punto 1.1.8 abbiamo visto che la posizione dei nodi fittizi nella partizione nodale estesa determina il tipo di curva approssimante. L'utente può scegliere se creare una traiettoria aperta (*Open*), interpolante agli estremi (*End points interpolant*, di default all'avvio) oppure chiusa (*Closed*). La figura 4.7 mostra le differenze a partire dagli stessi punti di controllo.

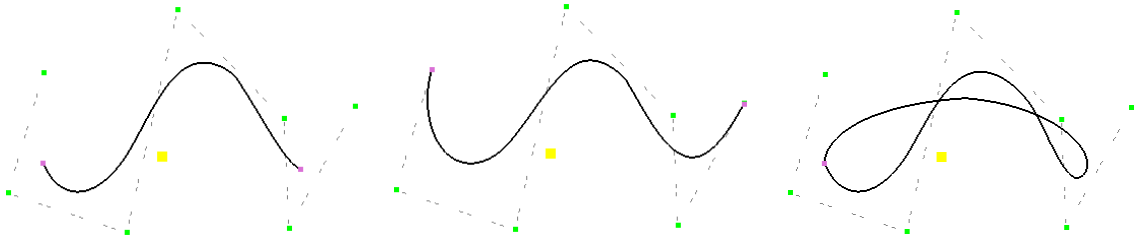


Figura 4.7: Curva aperta, interpolante agli estremi e chiusa dato lo stesso poligono di controllo

LS approximant

L'utente può anche scegliere di approssimare i punti dati nel senso dei minimi quadrati tramite il sottomenù *LS approximant*.

Selezionando l'opzione *Spline* si chiede al software di approssimare la nuvola di punti servendosi della base delle B-spline. Chiarimenti riguardo questo metodo sono disponibili al punto 1.3.2. È altresì possibile un'approssimazione simile ma servendosi della base monomiale, che va a formare la cosiddetta matrice di Vandermonde. Il nome dato a questa opzione è *Polynomial* e l'utente può vedere facilmente la differenza tra le due approssimazioni nella figura 4.8.

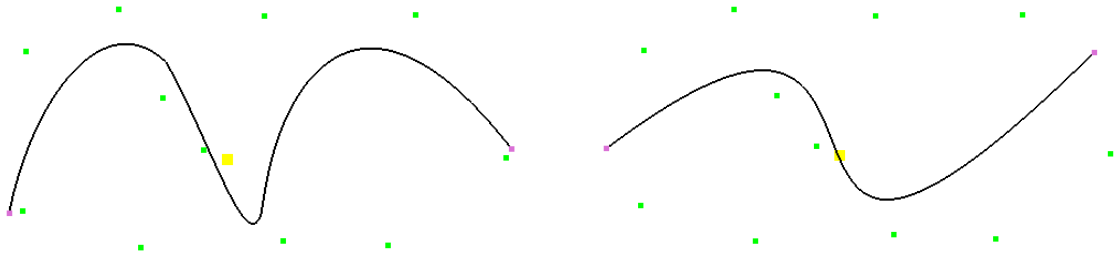
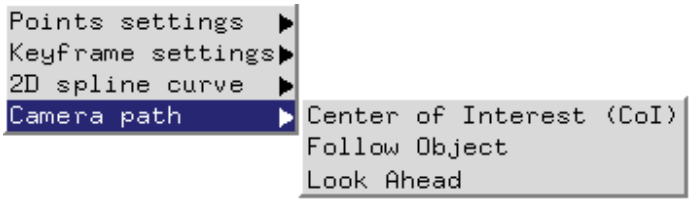


Figura 4.8: Approssimazione nel senso dei minimi quadrati tramite spline di ordine 4 (a sinistra) e polinomio di ordine 5 (a destra)

Si noti che non viene più disegnato il poligono di controllo. Le due curve approssimano infatti una nuvola di punti in successione che non hanno alcun legame fra loro. La spline generata tramite la base delle B-spline è formata da tanti polinomi di ordine 4 raccordati agli estremi. La base monomiale genera invece un unico polinomio di ordine pari alla metà dei punti dei punti dati. In entrambi i casi l'utente deve inserire minimo 10 punti.

Camera path type

Il sottomenù *Camera path type* permette all'utente di variare il tipo di obiettivo della telecamera. Una panoramica su tali tipi è presente al punto 2.3.4.



Il software è impostato inizialmente sulla modalità *Center of Interest (CoI)*. Questo significa che l'obiettivo consiste in uno (o più) oggetti della scena, scelti tramite keyframe nella Scene Window.

La modalità *Follow Object* sposta l'orientamento della telecamera verso un piccolo aeroplano di carta posto su un'altra curva. Nel momento in cui questa opzione viene selezionata potremmo creare una nuova curva (di colore violetto) tramite punti di controllo. Tutte le opzioni dei sottomenù *Control points settings* e *Approximant spline settings* avranno come nuovo riferimento la curva sulla quale si muove il nostro obiettivo. In questo modo l'utente può avere massimo controllo anche su tale percorso. Selezionando un altro tipo di orientamento il focus delle suddette operazioni tornerà sulla curva della telecamera.

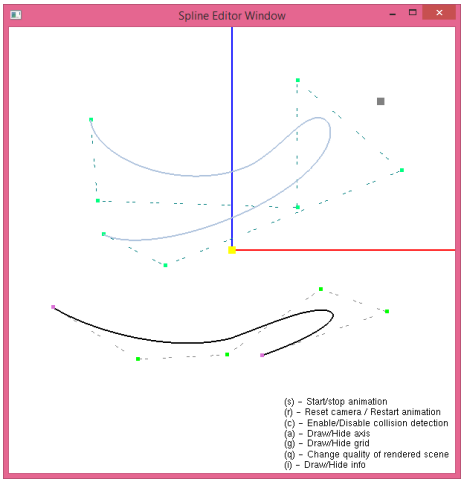


Figura 4.9: Modalità Follow Object: percorso della camera (in nero) e percorso dell'oggetto obiettivo (in violetto)

L'ultima modalità disponibile è la *Look Ahead* che prevede l'obiettivo posto sulla stessa curva della telecamera ad una certa distanza fissa davanti ad essa. La posizione dell'aeroplano apparirà come un punto di colore blu.

Comandi e informazioni

La figura 4.10 mostra i principali comandi disponibili per la Spline Editor Window. Questi possono essere visualizzabili nella parte in basso a destra della finestra. Esaminiamoli in ordine.

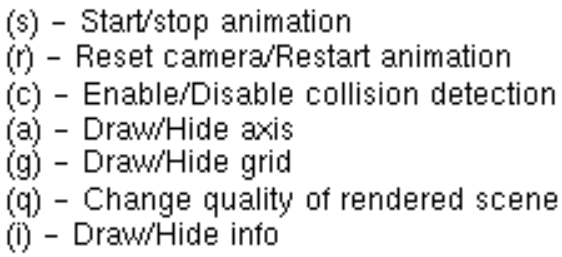


Figura 4.10: Comandi principali

Il tasto [s] fa partire o arresta l'animazione di camera motion. L'utente potrà vedere la telecamera (in grigio) spostarsi lungo la curva alla velocità specificata dalla Speed Control Window. In ogni momento ciò che sarà ripreso verrà mostrato sulla Scene Window.
La telecamera può essere riportante alla posizione originaria tramite il tasto [r]. Se premuto durante l'animazione questa ripartirà dall'inizio.
Il controllo sulle collisioni può essere disabilitato premendo il tasto [c]. Di default le collisioni tra percorso e mesh vengono sempre rilevate. In caso di possibile collisione rilevata tramite *bounding box* (si rimanda al punto 3.4.1 per maggiori informazioni) verrà mostrato un messaggio di errore nella parte alta della finestra. La parte di curva che collide viene evidenziata con il colore rosso, come mostrato nella figura 4.11.

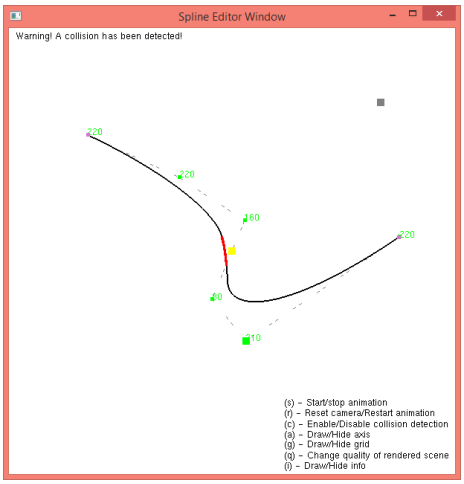


Figura 4.11: Avviso di collisione (notare la parte rossa della curva)

Il tasto [a] nasconde gli assi, mentre il tasto [g] mostra una griglia (di colore giallo) utile per inserire punti in posizione specifiche. I quadrati che si originano hanno un lato lungo 50.
Il tasto [q] permette di mostrare solo alcune parti della scena. Alla prima pressione saranno visibili solo la casetta e il faro, che sarà l'unica mesh rimasta se il tasto viene premuto la seconda volta.

Nel caso in cui l'utente voglia rapidamente vedere le configurazioni attuali del software può premere il tasto [i] per vedere le relative informazioni. Queste saranno visibile nella parte in basso a sinistra della finestra. La figura 4.12 ne mostra l'ingrandimento.

Objective: Center of Interest (Col)
Spline type: End points interpolant
Parameterization: Uniform
Mode: Insert a new point
Collision: Enabled
Animation: Idle
Quality: Draw all
Spline and points in 3D window: Visible

Figura 4.12: Informazioni sullo stato corrente

4.4.2 Scene Window

La finestra più grande è quella che contiene la scena ripresa dalla telecamera. Questa è in tre dimensioni e fa uso della prospettiva. Agli assi X e Z si aggiunge Y, di colore verde. Non appena si fa attua un qualsiasi cambiamento alla curva nella Spline Editor Window questo è subito visibile nella finestra della scena. Per validare la camera motion è stata costruita una scena 3D apposita descritta nel capitolo precedente. Una qualsiasi altra scena 3D potrebbe essere utilizzata all'interno di CaMot.

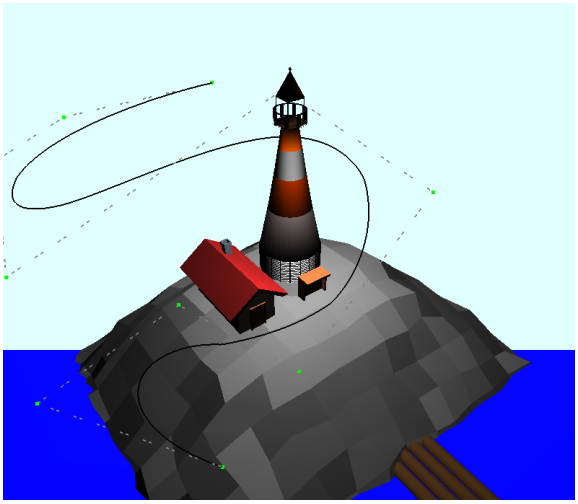


Figura 4.13: Esempio di traiettoria seguita dalla telecamera visibile nella scena 3D

La figura 4.13 mostra come appare il percorso della telecamera non appena vengono inseriti i punti di controllo necessari per definirlo.
Le seguenti immagini mostrano invece un esempio di camera motion nelle modalità Look Ahead (figura 4.14) e Follow Object (figura 4.15).

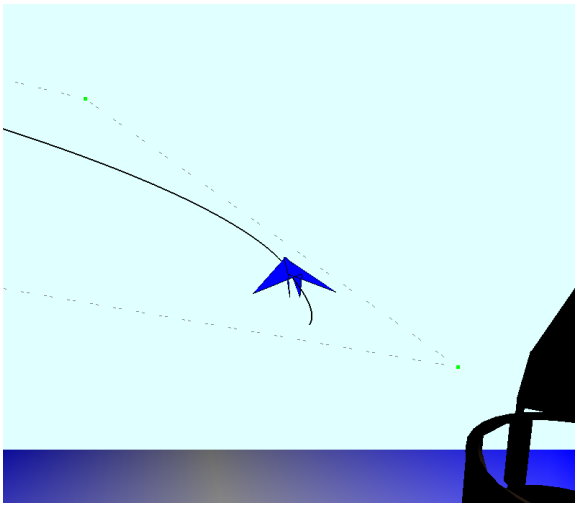


Figura 4.14: Modalità Look Ahead: l'oggetto obiettivo è sulla stessa traiettoria della telecamera

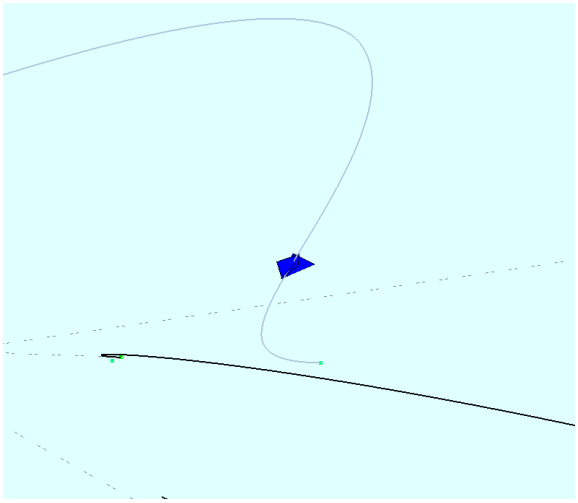


Figura 4.15: Modalità Follow Object: l'oggetto obiettivo è su una traiettoria diversa

Picking e keyframe

L'utente ha la possibilità di spostare l'orientamento della telecamera verso uno o più oggetti della scena in ogni momento. Per farlo è sufficiente un click con il tasto sinistro del mouse su una mesh

(*picking*). Se questa è stata rilevata con successo essa si colorerà di giallo. A questo punto l'utente può aprire un menù (figura 4.16) tramite tasto destro.

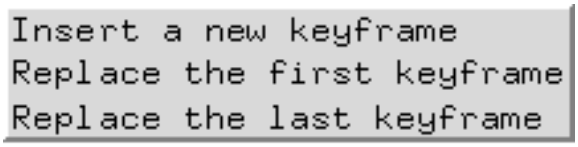


Figura 4.16: Menù per inserimento di un keyframe

Verranno presentate tre opzioni relative alla posizione del keyframe da creare. Questo obbligherà la telecamera ad orientarsi verso la mesh selezionata quando, durante il suo movimento sul percorso, si troverà su tale punto. Questo meccanismo è spiegato nel dettaglio al punto 2.3.5.
L'opzione *Insert a new keyframe* posizionerà il keyframe esattamente nel mezzo della curva. Questo sarà immediatamente visibile come un punto di colore viola nella Spline Editor Window. La sottosezione precedente spiega come spostarne eventualmente la posizione tramite tastierino numerico.
Le rimanenti opzioni *Replace the first keyframe* e *Replace the last keyframe* rimpiazzano rispettivamente l'orientamento iniziale e finale della telecamera.

Traslazione e rotazione della camera in scena

I comandi riservati alla Scene Window sono principalmente relativi al movimento della telecamera fuori dall'animazione. Essi sono tutti registrati sul tastierino numerico secondo tale schema:

- I tasti [4] e [6] muovono la telecamera rispettivamente in indietro e in avanti lungo l'asse X.
- I tasti [2] e [8] muovono la telecamera rispettivamente in indietro e in avanti lungo l'asse Y.
- I tasti [0] e [5] muovono la telecamera rispettivamente in indietro e in avanti lungo l'asse Z.

Le modifiche delle componenti X e Z saranno immediatamente visibile nella Spline Editor Window. All'avvio l'orientamento della telecamera fuori dall'animazione sarà sempre fisso sull'origine degli assi. Per poterlo spostare è sufficiente premere il tasto [o] e successivamente i tasti per il movimento sopra riportati.
Se l'utente desidera nascondere la curva spline e il suo poligono di controllo basterà premere il tasto [h].

4.4.3 Speed Control Window

La finestra dalle dimensioni più piccole è quella che permette il controllo della velocità. Al suo interno è visibile un grafico (tempo-spazio) che indica la posizione della telecamera in relazione alla percentuale di curva percorsa al tempo t. La sezione 2.2 spiega nel dettaglio il funzionamento.

All'interno di tale finestra si trova la funzione che controlla la velocità della camera. Questa parte sempre dall'origine degli assi e termina nell'angolo in alto a destra del quadrato, coincidente con il termine della curva che la telecamera deve raggiungere alla fine dell'animazione. La pendenza della curva in un dato istante misura la velocità di spostamento, mentre la sua monotonia (l'andamento di crescita o decrescita della funzione) misura la direzione verso cui la telecamera si muove (in avanti se la funzione è crescente in tale istante, all'indietro altrimenti). Se la curva per un certo intervallo di tempo è piatta significa che la telecamera rimane immobile.
La funzione della velocità può essere modificata tramite appositi keyframe (di colore rosso). Questi punti sono obbligatoriamente interpolati dalla curva e possono essere spostati a piacimento dall'utente. Devono sempre essere presenti almeno due keyframe, in quanto la funzione interpolatrice è una spline di ordine 4 che per poter essere definita ha bisogno di almeno altri due punti interposti tra l'origine degli assi e l'angolo in alto a destra del quadrato.

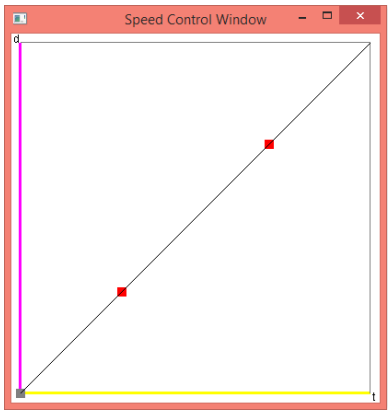


Figura 4.17: Speed Control Window all'avvio

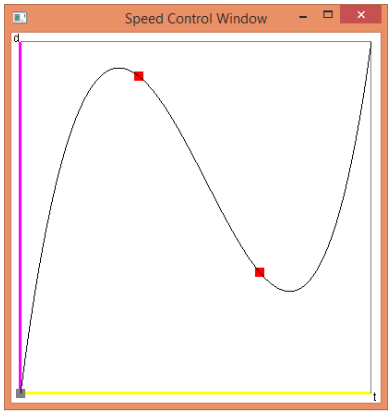


Figura 4.18: Controllo tramite keyframe

La figura 4.17 mostra come si presenta la finestra del controllo della velocità all'avvio del software. La curva è inizialmente un segmento che interpola due keyframe allineati con gli estremi della curva. La camera si muoverà a velocità costante lungo la propria traiettoria parametrizzata alla lunghezza d'arco. Cliccando sui keyframe e trascinandoli possiamo dare alla curva una nuova forma e modificare l'andamento della camera. La figura 4.18 è un esempio di controllo della velocità tale da permettere alla curva di arrivare rapidamente a completare circa il 90% del percorso, per poi tornare indietro fino a circa il 30% della suddetta e infine accelerare velocemente fino al completamento.
L'utente può inserire a piacere nuovi keyframe al fine di avere maggiore libertà sulla forma della curva. Tramite click con il tasto destro viene mostrato un semplice menù (figura 4.19) con le voci *Add a point* e *Delete last point*. La prima permette semplicemente di aggiungere un keyframe al centro della curva di controllo mentre il secondo permette l'eliminazione dell'ultimo keyframe

inserito. Quest'ultima operazione non avrà effetto se il numero di keyframe non estremi è pari a due.

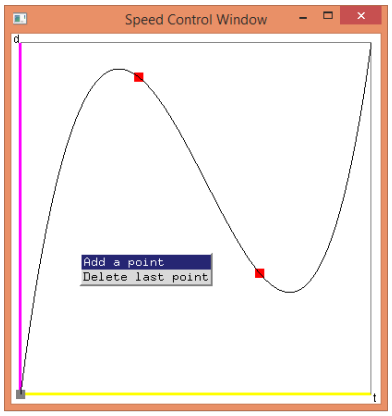


Figura 4.19: Menù per inserire o eliminare i keyframe

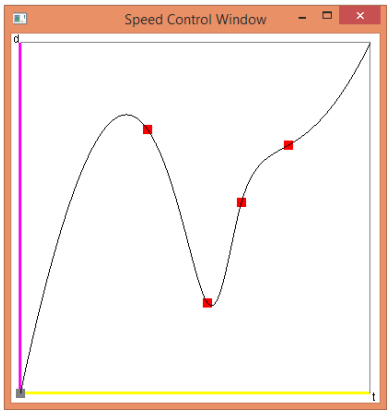


Figura 4.20: Curva con 4 keyframe

Durante l'animazione l'utente potrà osservare l'andamento della velocità della camera (in grigio) ad ogni singolo frame.

Bibliografia e sitografia

In questo capitolo sono citati i libri, le pubblicazioni e le risorse online che mi sono state utili per la realizzazione del progetto e di questa tesi. A questi vanno aggiunte le dispense fornitemi dalle professoresse Serena Morigi e Damiana Lazzaro. Le immagini sono reperibili tramite qualsiasi motore di ricerca.

Bibliografia

[1] Carl R. de Boor. *A Practical Guide to Splines, Revised Edition*. 2003.

[2] Rick Parent. *Computer Animation: Algorithms and Techniques*. 2002.

[3] Ken Shoemake. *Animating Rotation with Quaternion Curves*. 1985.

[4] Carl R. de Boor e John R. Rice. *Least squares cubic spline approximation I - Fixed Knots*. 1968.

[5] David L. B. Jupp. *Approximation to Data by Splines with Free Knots*. 1978.

[6] Gerald Farin e Dianne Hansford. *The Essentials of CAGD*. 2000.

[7] Stefano Beccaletto. *Leap Aided Modelling & Animation*. 2017.

Sitografia

[8] Microsoft Visual Studio. URL: `https://www.visualstudio.com/`.

[9] Blender. URL: `https://www.blender.org/`.

[10] OpenGL. URL: `https://www.opengl.org/`.

[11] FreeGLUT. URL: `http://freeglut.sourceforge.net/`.

[12] Gnu Scientific Library (GSL). URL: `https://www.gnu.org/software/gsl/`.

[13] *Spline Approximation of Functions and Data*. URL: http://folk.uio.no/in329/nchap5.pdf.

[14] Martin John Baker. *Euclidean Space*. URL: http://www.euclideanspace.com/maths/algebra/.

[15] Christopher Twigg. *Camera movement along a spline*. 1985. URL: http://www.cs.cmu.edu/~462/projects/assn2/assn2/cameraMovement.pdf.

[16] Jim Armstrong. *Arc-Length Parameterization Part I*. 2006. URL: http://algorithmist.net/docs/arcparam.pdf.

[17] John W. Peterson. *Arc Length Parameterization of Spline Curves*. URL: http://kalyaev.com/2010/20100303/RE-PARAM.PDF.

[18] B. Stenseth. *Frames*. 2009. URL: http://www.it.hiof.no/~borres/j3d/explain/frames/p-frames.html.

Conclusioni

Con il progetto CaMoT si è realizzato un basilare motore grafico per gestire animazioni di camera motion. È mio sincero auspicio che questa tesi possa in futuro aiutare chi come me vorrebbe cimentarsi in un simile lavoro. Ne riassumo brevemente i contenuti.
Il capitolo 1 descrive la parte matematica alla base delle curve spline usate per creare la traiettoria della telecamera e dà soluzioni ai problemi di interpolazione e approssimazione di punti. Nel capitolo 2 ho discusso delle tre componenti principali di una camera motion e in particolare mi sono soffermato sulle tecniche più comuni riguardanti la gestione della velocità e dell'orientamento della camera durante il suo movimento. Il capitolo 3 descrive la parte di modellazione e di caricamento delle mesh realizzate con Blender nella scena 3D. Ho inoltre scritto una breve spiegazione circa l'utilizzo delle bounding box e della bounding volume hierarchy per controllare eventuali collisioni. Il capitolo 4 si concentra infine sulla parte implementativa del software CaMoT e ne descrive l'interfaccia utente, le funzionalità offerte e le tecnologie utilizzate.
Il progetto si presta ad essere ulteriormente arricchito in diversi aspetti quali l'integrazione di device per il riconoscimento automatico di gesture e l'estensione a più tipologie di animazione lungo il percorso.

Ringraziamenti

Desidero ringraziare principalmente le professoresse Serena Morigi e Damiana Lazzaro per l'immenso aiuto che mi hanno dato nella realizzazione di questa tesi e per la continua disponibilità che mi hanno dimostrato durante i tanti ricevimenti. Il loro aiuto è stato determinante per consentirmi di superare difficoltà che parevano inizialmente insormontabili. A loro va tutta la mia stima. Ringrazio inoltre con tutto il cuore gli amici - vecchi e nuovi - che mi hanno accompagnato durante questi tre anni. Sono senza dubbio convinto che senza il loro appoggio sarebbe stato molto più arduo arrivare a questo successo. Le esperienze vissute insieme saranno difficilmente dimenticabili e non posso non augurare loro di raggiungere qualsiasi obiettivo abbiano in mente nello studio, nel lavoro e nella vita.