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

CORSO DI LAUREA IN SCIENZE E TECNOLOGIE INFORMATICHE

UNITY3D E SPLINE: SVILUPPO DI UN VIDEOGAME BASATO SULLE CURVE SPLINE

Relazione finale in
Metodi numerici per la grafica

Relatore
Damiana Lazzaro

Presentata da
Pierluigi Montagna

II Sessione
Anno Accademico 2012 – 2013

Indice

Introduzione

Nella tesi in questione si vogliono analizzare e descrivere metodi e strumenti utilizzati per la realizzazione di un videogame che implementa come meccanica di gioco le curve Spline.

Il progetto consiste in un *puzzle game*, un genere che enfatizza la risoluzione di enigmi tramite l'utilizzo della logica. Nello specifico, l'utente avanzerà di livello in livello descrivendo dei percorsi tramite dei punti di controllo e raggiungendo degli obiettivi.

Il contesto della tesi è quello dello sviluppo di applicazioni grafiche interattive. Si parlerà di come si implementano determinate tecniche per la progettazione di un videogioco basato su un motore grafico 3D. Un motore 3D (*real time 3D engine*) è un software progettato per rappresentare su superfici 2D, come uno schermo, una scena composta da diversi elementi a tre dimensioni.

Per poter descrivere il funzionamento del progetto la tesi tratterà sia della teoria delle curve, cercando di spiegare come è possibile descrivere dei percorsi nel calcolatore, giustificando per quale motivo sono stati scelti determinati algoritmi, e sia di quali strumenti sono stati utilizzati per la creazione del videogame, soffermandosi sul funzionamento dell'engine e fornendo informazioni sull'implementazione del codice.

Non saranno escluse dalla tesi informazioni riguardanti lo sviluppo dell'idea e del lato artistico di un videogame, anche se questa parte sarà più discorsiva essendo l'argomento qualcosa che non si rifà a dei criteri ben precisi, ma piuttosto ai gusti e alle capacità dello sviluppatore.

Il videogioco si mostra con una grafica in 2D ed è giocabile sia su PC con sistema operativo Windows che su mobile con sistema operativo Android. Il nome scelto per l'applicazione è Rocket Spline in base alla particolarità del

gameplay, che è quella di dover muovere un razzo lungo delle curve spline, descritte dal giocatore mediante l'inserimento di vertici di controllo.
L'ambiente di sviluppo utilizzato è Unity3D, completamente dedicato alla programmazione di videogiochi, con un buon render grafico e supporto multipiattaforma. Nella tesi si vedranno parecchi esempi di codice scritti appositamente per tale engine, tuttavia diverse teorie e approcci sono comuni nell'ambito del game programmig e possono essere adattati anche a diversi ambienti di sviluppo.
Rocket Spline è basato sull'idea di un gioco casual, il cui obiettivo è quello d'intrattenere l'utente con una serie di rompicapo la cui difficoltà aumenterà con l'avanzare del gioco. Avanzando nel gioco l'utente migliorerà nel controllare la curva e magari imparerà, anche se inconsciamente, alcune proprietà delle curve spline.
La tesi è così raggruppata: il capitolo 1 tratta la teoria delle curve spline; partendo dai fondamenti si dimostra come sia possibile definire matematicamente una curva sia per interpolazione che per approssimazione. Gli argomenti trattati includono la parametrizzazione, le basi b-spline, l'algoritmo di De Boor e le proprietà delle curve spline.
Nel capitolo 2 si discute dell'ambiente di sviluppo utilizzato per la creazione del progetto. Si parla del motore grafico di Unity3D e si spiega qual è la filosofia dietro al suo linguaggio di programmazione, descrivendone gli elementi essenziali.
Nel capitolo 3 si illustrano i processi che hanno portato alla creazione di Rocket Spline, lo sviluppo dell'idea e del codice. Nello specifico si espone in che modo siano stati gestiti gli input sia per la versione mobile che per quella pc e si spiega come sono stati gestiti gli eventi all'interno del gioco. Parte di questo capitolo tratta anche l'interfaccia grafica e l'implementazione delle curve spline all'interno del progetto.

Il capitolo 4 parla del lato artistico del videogame, spiegando come è stata creata la grafica e il sonoro.

Capitolo 1

Curve nel calcolatore

In matematica una curva è una figura geometrica continua come ad esempio una retta o un ellisse. Le curve sono spesso utilizzate per definire oggetti e forme e sono ampiamente utilizzate in diversi programmi di grafica. In questo paragrafo si vogliono descrivere quali siano le tecniche matematiche utilizzate per poter definire curve nel piano, in particolare facendo riferimento a quelle che ne permettono una buona resa nel calcolatore grazie ad algoritmi computazionalmente veloci. Nel computer è spesso indispensabile dover definire una curva senza doverne memorizzare tutti i suoi valori in memoria. A tale proposito si è pensato di descrivere delle figure semplici attraverso delle funzioni matematiche ($y = f(x)$), tuttavia in questo modo non è possibile poter definire delle figure come una circonferenza, poiché ad ogni valore del dominio è associato uno ed un solo valore del codominio. Si potrebbe considerare l'equazione implicita $f(x,y) = 0$, la quale però renderebbe solo più complessa la descrizione di determinate curve.

Si introduce il concetto di *forma parametrica* che descrive una curva nel piano tramite l'utilizzo di una coppia di funzioni in cui una rappresenta l'ascissa e l'altra l'ordinata in base alla variazione del valore di un parametro comune.

$$x = x(t),\ y = y(t) \quad \text{con } t \epsilon [t_{min}, t_{max}]$$

Formare delle curve attraverso delle funzioni non basterebbe per fornire un metodo intuitivo per poterle disegnare sul piano, si vuole trovare un metodo che data una serie di punti crei una curva. Seguendo questo principio si pensi di voler definire la curva tramite una sua discretizzazione: dati diversi punti nel piano P_1, P_2, P_3, ecc la si vuole formare interpolando o

approssimando questi valori; basterebbe assegnare ad ogni punto un parametro t_1, t_2, t_3,ecc e poi risolvere il rispettivo problema di interpolazione o approssimazione per le funzioni *x(t)* e *y(t)*.
A partire dalla posizione dei punti P_1, P_2, P_3, ecc si possono ottenere diverse curve, la cui forma dipenderà dalle funzioni base utilizzate per risolvere il problema di interpolazione/approssimazione per le due componenti parametriche, tra queste si ricordano le B-spline, le funzioni base di Hermite e di Bezier. Le curve create attraverso le funzioni base di Hermite hanno un metodo di definizione che differisce da quello pensato per il progetto, infatti con Hermite si risolve un problema in interpolazione, in cui la curva passa per ogni nodo, cosa che renderebbe il gameplay troppo semplice, diversamente si è pensato a qualcosa più simile a quello che accade con le curve di Bezier, in cui la funzione polinomiale interpola solo il primo e l'ultimo punto, approssimando il suo andamento in base alla posizione dei nodi intermedi. La stessa metodologia è applicata anche per funzioni spline a nodi multipli basate su B-spline, che però risultano facili da implementare e godono di proprietà che garantiscono un migliore controllo sulla forma e per questo sono state scelte per il progetto.

Curve Spline a nodi semplici

Una spline è una funzione polinomiale a tratti con una maggiore flessibilità nei punti in cui i pezzi della curva si collegano, i cosiddetti knots .[1]
Nell'ambito della risoluzione di problemi di interpolazione le spline sono abbastanza popolari, poiché permettono di evitare il fenomeno di Runge, ovvero l'oscillazione incontrollata della curva agli estremi, grazie alla loro capacità di poter interpolare utilizzando polinomi di grado basso.

Nella computer grafica invece sono ampiamente conosciute, grazie alla loro semplicità nel definire curve e la capacità di approssimazione di forme complesse. Nel seguente paragrafo si cercherà di introdurre i concetti fondamentali per comprendere il funzionamento delle spline, quali quello di partizione e B-spline, comprese le tecniche matematiche e gli algoritmi che ne permettono la loro valutazione.

Definizione di spline polinomiale a nodi semplici

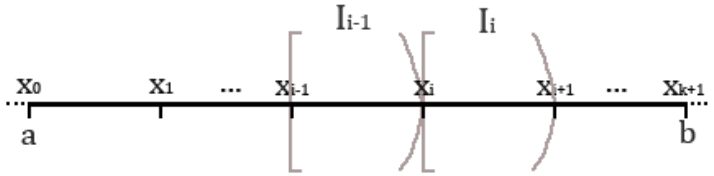
Sia [a,b] un intervallo chiuso e limitato dell'asse reale e sia Δ una partizione così definita:

$$\Delta = \{a = x_0 < x_1 < \ldots < x_k < x_{k+1} = b\}$$

che a sua volta genera k+1 sotto intervalli:

$$I_i = [x_i , x_{i+1}) \; i=0, \ldots k\text{-}1$$

$$I_k = [x_k , x_{k+1}]$$



Fissato un intero positivo m < k, si definisce spline polinomiale di ordine m e grado m-1, una funzione s(x) tale che in ciascun intervallo I_i con i = 0, ...k coincida con un polinomio di ordine m $s_i(x)$, e che soddisfa le seguenti proprietà di raccordo per ogni nodo semplice della spline.

$$\frac{d^j s_{i-1}(x_i)}{dx^j} = \frac{d^j s_i(x_i)}{dx^j} \quad i=1,\ldots k \text{ e } j=0,\ldots m\text{-}2$$

Ovvero si impone che per ogni punto $x_1, x_2, \ldots x_k$ le funzioni $s_i(x)$ coincidano in valore e ogni derivata di grado compreso tra 1 e m-2, in questo modo si

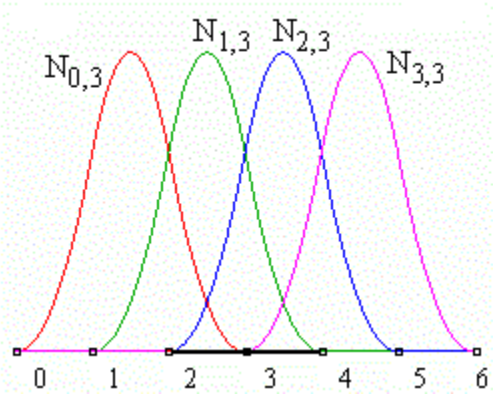
ha che s(x) è continua nell'intervallo [a,b] insieme alle sue derivate fino a quella di ordine m-2, si dice che $s(x) \in C^{m-2}_{[a.b]}$.

Lo spazio delle spline di ordine m viene solitamente indicato con Sm(Δ), dove Δ è la partizione nodale precedentemente definita, la sua dimensione è pari a m + k che risulta essere il numero di polinomi necessari per poter valutare la spline (di ordine m) in un punto.

Funzioni base B-spline

Per calcolare efficientemente una funzione spline si utilizzano le basi B-spline, le cui caratteristiche ci permettono di valutare il polinomio tramite algoritmi ottimi sia per la loro stabilità che dal punto di vista computazionale. Si definisce B-spline normalizzata di ordine m relativa al nodo x_i una funzione $N_{i,m}(x)$ che gode delle seguenti proprietà:

- Proprietà del supporto compatto: $N_{i,m}(x) = 0$ per $x < x_i , x > x_{i+m}$
- Condizione di normalizzazione: $\int_{xi}^{x_{i+m}} N_{i,m}(x) = (\frac{x_{i+m} - x_i}{m})$
- Coincide in ogni intervallo I_j con j=i,...m-1 con un plinomio di ordine m che si raccorda opportunatamente con i vicini permettendo che $N_{i,m}(x) \in C^{m-1}_{[a,b]}$.



In figura un esempio di B-spline con m = 3 definite per i=0,...3.

Per valutare le funzioni base si utilizzano le formule ricorsive di Cox, le quali permettono di calcolare $N_{i,m}(x)$ come combinazione lineare di due funzioni di ordine inferiore:

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

$$N_{i,h}(x) = \left(\frac{x - x_i}{x_{i+h-1} - x_i} N_{i,h-1}(x) + \frac{x_{i+h} - x}{x_{i+h} - x_{i+1}} N_{i+1,h-1}(x)\right) \text{ con } h=2,\ldots m$$

Grazie a Cox è possibile immaginare già un primo algoritmo di valutazione di una B-spline definita su x_i di ordine m, si applica ricorsivamente la formula moltiplicando le due funzioni di grado inferiore per il rispettivo supporto, fino a raggiungere la condizione base per cui h = 1.

Partizione nodale estesa

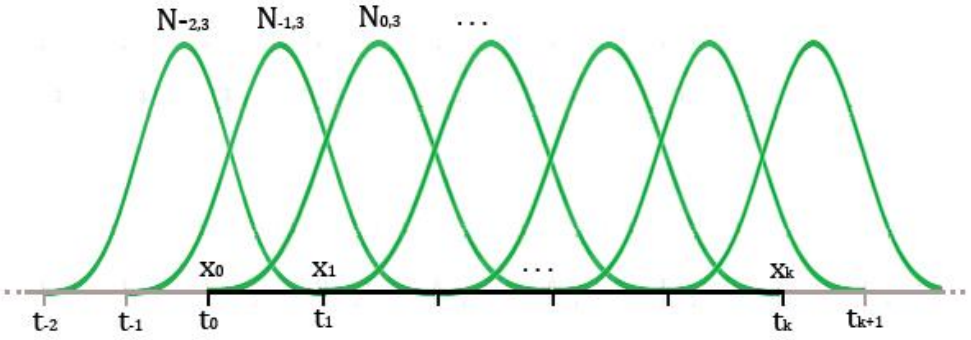
Come abbiamo precedentemente enunciato $S_m(\Delta)$, ovvero lo spazio delle spline di ordine *m*, ha dimensione *m* + *k*, tuttavia la partizione nodale non contiene nodi a sufficienza per poter costruire tale numero di funzioni base B-spline. È necessario introdurre il concetto di partizione nodale estesa Δ^* in cui sono inseriti *2m* nodi fittizi distribuiti equamente agli estremi della partizione semplice, oppure coincidenti rispettivamente con il primo e l'ultimo estremo dell'intervallo.

Definiamo $\Delta^* = \{ t_i \}_{i=-m+1}^{m+k}$ come:

$$t_i \leq a \quad i=-m+1, \ldots , 0 \text{ con } t_0 \equiv a$$

$$t_i \equiv x_i \quad i=1, \ldots, k$$

$$t_i \geq b \quad i=k+1, \ldots, k+m \text{ con } t_{k+1} \equiv b$$



Mediante questa partizione è ora possibile definire le m+k B-spline normalizzate di ordine m tramite la formula di Cox.

Si dimostra che l'insieme delle $N_{i,m}(x)$ con i=-m+1,…,k costituisce una base per $S_m(\Delta^*)$. A questo punto possiamo esprimere la nostra spline $s(x) \in S_m(\Delta^*)$ come:

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

Inoltre per la proprietà del supporto compatto delle B-spline sappiamo che per un $x \in [t_l, t_{l+1})$ le funzioni base diverse da zero sono le m-1 precedenti il punto t_l, l'ultima formula può essere scritta come:

$$s(x) = \sum_{i=l-m+1}^{l} \alpha_i N_{i,m}(x)$$

Risoluzione del problema di interpolazione

Risulta essere interessante arrivati a questo punto conoscere come poter risolvere il problema di interpolazione dei punti per ottenere la curva. Spieghiamo come è possibile interpolare i nodi semplici (x_i, y_i), i=0,…N+1, ricordando che bisogna risolvere due problemi, uno per ogni componente parametrica. Poiché la funzione $s(x) \in S_m(\Delta)$ dipende dalla posizione e dal

numero di nodi che la caratterizzano, è necessario definire come posizionare i nodi veri della partizione:

$$\Delta = \{a < t_1 < \ldots < t_k < b\}$$

Si tratta di risolvere un sistema lineare in cui le incognite solo i valori di α_i, si impone:

$$s(x_i) = y_i \qquad i=0,\ldots,N+1$$

Se lo spazio delle spline di ordine m=4 ha dimensione N+2, cioè N+2-m=N-2 nodi il sistema lineare che si ottiene è il seguente:

$$\begin{cases} N_{1,4}(x_0)\alpha_1 + N_{2,4}(x_0)\alpha_2 + \cdots + N_{N+2,4}(x_0)\alpha_{N+2} = y_0 \\ N_{1,4}(x_1)\alpha_1 + N_{2,4}(x_1)\alpha_2 + \cdots + N_{N+2,4}(x_1)\alpha_{N+2} = y_1 \\ \ldots \\ N_{1,4}(x_{N+1})\alpha_1 + N_{2,4}(x_{N+1})\alpha_2 + \cdots + N_{N+2,4}(x_{N+1})\alpha_{N+2} = y_{N+1} \end{cases}$$

Tale sistema risulta quadrato e potrebbe essere risolubile, non possiamo dire nulla sul suo rango poiché alcune configurazioni di nodi potrebbero avere delle colonne tutte uguali a 0.

Il teorema di Schoenberg-Whitney ci fornisce una condizione necessaria e sufficiente che ci permette di affermare che il sistema ha rango massimo.

Teorema:

Sia assegnata una successione di nodi $t_1 < \ldots < t_{s+m}$ e una successione di punti di osservazione $\xi_1 < \ldots < \xi_s$ e definita la successione di b-spline $N_{1,m}(x), \ldots, N_{s,m}(x)$. Il sistema lineare che interpola i punti $(\xi_i, f(\xi_i))$ con i=1,…,s è il seguente:

$$\sum_{j=1}^{s} \alpha_j N_{j,m}(\xi_i) = f(\xi_i) \;\; j = 1, \ldots, s$$

Da cui risulta la seguente matrice quadrata:

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

La matrice A ha rango massimo se e solo se $t_i < \xi_i < t_{i+m}$ per i=1,...,s, ovvero se per ciascuna funzione b-spline esiste almeno un punto di interpolazione presente nel suo intervallo.

Leghiamo quindi la scelta dei nodi ai punti di interpolazione secondo Schoenberg.

In questa possibile configurazione scartiamo i due punti immediatamente più interni rispetto agli estremi:

k=N-2 $t_0 \equiv x_0, t_1 \equiv x_2, \ldots, t_k \equiv x_{N-1}, t_{k+1} \equiv x_{N+2}$

Da cui notiamo che $\dim\big(S(\Delta^*)\big) = k + m = N - 2 + m$, con m=4 otteniamo come risultato N+2 incognite per N+2 condizioni di interpolazione ai punti.

Questo garantisce che la matrice delle funzioni base sia a rango massimo, sappiamo quali valori di t_i scegliere per poter costruire la nostra spline mediante l'interpolazione dei punti $(\xi_i, f(\xi_i))$, risolvendo il sistema si ottengono N+2 equazioni ognuna delle quali ha N+m incognite.

Un'altra possibile scelta sarebbe potuta essere quella di porre k=N prendendo tutti nodi, si ha $\dim\big(S(\Delta^*)\big) = k + m = N + m$, caso in cui m=4 si hanno N+2 condizioni e N+4 incognite, per ottenere un sistema quadrato si aggiungono due altre condizioni, solitamente si utilizzano condizioni ai bordi come quella della spline cubica naturale in cui si impone nullo il valore della derivata seconda ai bordi.

In conclusione esprimiamo la curva spline interpolante come:

$$C(t) = \begin{cases} x(t) = \sum_{i=-m+1}^{N} \alpha_i N_{i,m}(t) \\ y(t) = \sum_{i=-m+1}^{N} \beta_i N_{i,m}(t) \end{cases}$$

Dove α_i i=-m+1,...,N sono la soluzione del sistema relativo alla funzione parametrica x(t) e delle opportune condizioni ai bordi, mentre β_i i=-m+1,...,N sono la soluzione del sistema relativo alla funzione parametrica y(t) e delle eventuali condizioni ai bordi.

Parametrizzazione

Nel precedente paragrafo abbiamo parlato di funzione parametrica senza preoccuparci di come vengono scelti i valori delle variabili t_i (dove i è l'indice dei punti). La scelta del modo in cui è calcolato il parametro t delle N+2 coppie relative ai punti che descrivono la curva è chiamata parametrizzazione, di seguito daremo una panoramica delle tecniche di parametrizzazione utilizzate nel calcolatore.

Parametrizzazione uniforme

In questo tipo di parametrizzazione i valori del parametro t sono scelti in modo che siano distribuiti equidistanti tra di loro all'interno dell'intervallo di variabilità [0,a] (dove solitamente a è scelto come N+1).

$$t_i = i\frac{a}{N+1} \text{ per } i = 0, \dots, N+1$$

La parametrizzazione uniforme risulta essere la più veloce algoritmicamente, tuttavia questa non tiene conto della distanza reale dei punti, quindi a seconda di come sono distribuiti i punti e da come è settato l'incremento del parametro t nell'algoritmo di valutazione della curva si possono ottenere risultati poco soddisfacenti.

Parametrizzazione mediante corda

In questo caso invece si tiene conto della distanza geometrica dei punti che descrivono la curva, si mantengono quindi invariate le proporzioni, la

differenza tra intervalli successivi è proporzionale alla distanza dei punti ad essi relativi.

$$t_0 = 0$$

$$t_i = t_{i-1} + \left|\left|P_i - P_{i-1}\right|\right| = t_{i-1} + \sqrt{(x_i - x_{i-1})^2 - (y_i - y_{i-1})^2}$$

Geometricamente questa parametrizzazione può essere vista come un'approssimazione della lunghezza dell'arco e porta risultati migliori rispetto a quella uniforme.

Altre parametrizzazioni

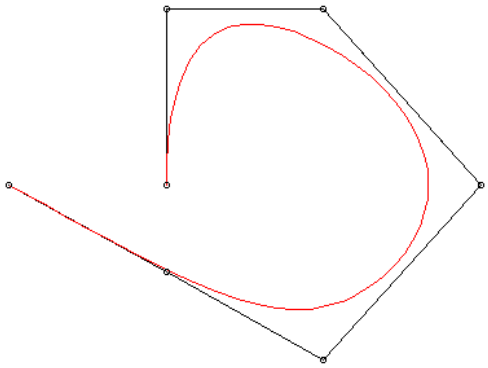
Le tecniche hanno un risultato migliore quando si utilizzano informazioni aggiuntive sulla curva, per esempio nella parametrizzazione di Foley oltre alla distanza tra i punti di valutazione della curva di tiene conto anche degli angoli formati dai segmenti ottenuti unendo i punti. Un'altra tecnica di parametrizzazione è quella centripeta in cui si considera una particolare distanza tra i punti, quella che nel modello fisico tiene anche conto della forza perpendicolare delle curve.

In generale non esiste una tecnica di parametrizzazione che è migliore delle altre, bisogna tenere conto di diversi fattori, in primis il costo computazionale dell'algoritmo utilizzato. Un'altra importante proprietà da tenere in considerazione è *l'invarianza per trasformazioni affini*, di questa proprietà gode la parametrizzazione uniforme, che non deve essere ricalcolata ogni volta che si hanno modifiche sulla curva.

Curve spline a nodi multipli

Vogliamo avvicinarci al concetto di costruire una curva per approssimazione della forma del poligono di controllo. Seguendo il modello

utilizzato dalle curve di Bezier: si immagini di avere dei segmenti che uniscono i nostri punti di valutazione, la figura ottenuta è detta *poligono di controllo* ed approssimerà la nostra funzione polinomiale.



A tale proposito non cercheremo più di risolvere un problema di interpolazione cercando di trovare i valori dei coefficienti da abbinare alle b-spline, utilizzeremo invece direttamente i valori come l'ascissa o l'ordinata relativa ai punti di controllo, come si vedrà in seguito. Questa curva prende il nome di curva B-spline, poiché il suo andamento è dettato principalmente dalle funzioni base omonime.

Le spline polinomiali a nodi semplici sono funzioni che presentano regolarità di classe C^{m-2} (dove m è l'ordine della spline), che è a volte troppo elevata per permettere alla curva di ottenere anche tratti irregolari, in cui ad esempio la funzione non è continua in derivata prima. Risulta essere importante poter aggiungere dei gradi di libertà alla spline, riducendo all'occorrenza le condizioni di raccordo sui punti, ciò può essere fatto introducendo il concetto di nodi multipli.[2]

Definizione di spline polinomiale a nodi multipli

Sia [a,b] un intervallo chiuso e limitato dell'asse reale e sia Δ una partizione così definita:

$$\Delta = \{a = x_0 < x_1 < \ldots < x_k < x_{k+1} = b\}$$

che a sua volta genera k+1 sotto intervalli:

$$I_i = [x_i , x_{i+1}) \; i=0, \ldots k\text{-}1$$

$$I_k = [x_k , x_{k+1}]$$

Dato un intero m maggiore di zero e il vettore $M = (m_1, m_2, \ldots, m_k)$ di interi positivi tali che $1 \leq m_i \leq m$ per i=1,…,k, si definisce spline polinomiale di ordine m con nodi $x_1, \ldots, x_k$ di rispettive molteplicità $m_1, \ldots, m_k$ una funzione $s(x)$ che in ciascun sotto intervallo I_i coincide con un polinomio $s_i(x)$ di ordine m e che inoltre soddisfi le seguenti condizioni di continuità nei nodi:

$$\frac{d^j s_{i-1}(x_i)}{dx^j} = \frac{d^j s_i(x_i)}{dx^j} \qquad i=1,\ldots k \text{ e } j=0,\ldots,m - mi - 1$$

La differenza con le spline a nodi semplici risiede nel fatto che tramite il vettore M si può adesso impostare un grado di libertà su uno specifico nodo, permettendo di variare il tipo di raccordo su quel punto, naturalmente minore sarà mi e maggiore sarà il numero di condizioni di raccordo per x_i. Notiamo come si ricade nel caso delle spline a nodi semplici quando $m_i = 1$ per ogni i=1,…,k.

Indichiamo con $S_m\ (\Delta, M)$ lo spazio delle spline polinomiali a nodi multipli x_i con rispettive molteplicità m_i per i=1,…,k, la sua dimensione risulta pari a m+K, dove $K = \sum_{i=1}^{k} m_i$.

Partizione nodale estesa per nodi multipli

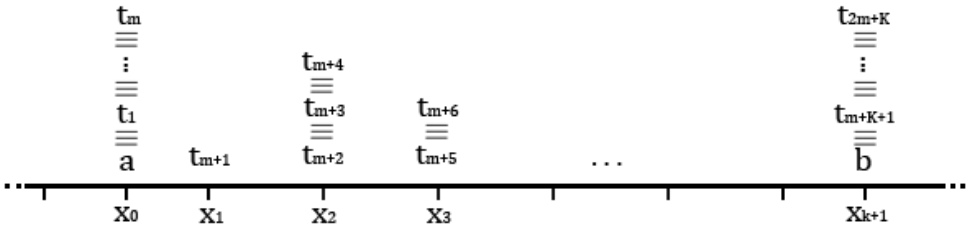
Analogamente per quanto avveniva per le spline a nodi semplici la partizione nodale formata dai soli k punti non è sufficiente per poter costruire tutte le spline presenti nello spazio di dimensione m + K, per questo motivo introduciamo la partizione nodale estesa $\Delta^* = \{t_i\}_{i=1}^{2m+K}$ fatta nel seguente modo su [a,b]:

- $t_1 \leq t_2 \leq \cdots \leq t_{2m+K}$ sono nodi semplici, ma possono coincidere sullo stesso punto xi.
- $t_m \equiv a, \qquad t_{m+K+1} \equiv b$

- I nodi interni $t_{m+1} \leq t_{m+2} \leq \cdots \leq t_{m+K}$ sono scelti in modo che coincidano m_i volte per ogni punti x_i con i=1,…,k.
- $t_j \leq a$ j=1,…,m-1
- $t_j \geq b$ j=m+K+2,…,2m+K

I punti x_i i=1,…,k sono anche chiamati breaking points, da cui partono tante funzioni base quanta è la sua molteplicità.

M=(1,3,2,...) m=4



Per formare $S_m(\Delta^*,M)$ si utilizzano le solite funzioni base b-spline, valutate tramite le formule ricorsive di Cox:

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

con h=2,…m

$$N_{i,h}(x) = \begin{cases} \left(\frac{x - t_i}{t_{i+h-1} - t_i} N_{i,h-1}(x) + \frac{t_{i+h} - x}{t_{i+h} - t_{i+1}} N_{i+1,h-1}(x) \right) se\ t_i \neq t_{i+h} \\ 0 \ altrimenti \end{cases}$$

Possiamo adesso quindi valutare la nostra funzione spline polinomiale a nodi multipli tramite la seguente formula:

$$s(x) = \sum_{i=1}^{m+K} c_i N_{i,m}(x) \quad x \in [a, b]$$

Proprietà delle B-spline a nodi multipli

Di seguito elenchiamo alcune proprietà legate alle b-spline che risultano utili alla comprensione dell'algoritmo utilizzato per la valutazione della curva.

- Non negatività: $N_{i,m}(x) \geq 0$ per $t \in [t_i, t_{i+m}]$
- Supporto locale (o compatto): $N_{i,m}(x) = 0$ $per\ t < t_i, t > t_{i+m}$ Partizione dell'unità: $\sum_{i=1}^{m+K} N_{i,m}(x) = 1$
- Formano una base per $S_m(\Delta^*, M)$.

Dal supporto locale possiamo restringere il campo di valori scanditi dalla sommatoria, sapendo di utilizzare solo le funzioni base il cui valore non è nullo, dato un $x \in [t_l, t_{l+1}]$ si ha:

$$s(x) = \sum_{i=l-m+1}^{l} c_i N_{i,m}(x)$$

Per le proprietà di non negatività e partizione dell'unità ci accorgiamo invece che la formula della spline è una combinazione lineare convessa dei coefficienti ci, da cui possiamo dire che:

$$\min\{c_i\} \leq s(x) \leq \max\{c_i\}$$

Ovvero che la nostra funzione spline sarà compresa tra il minimo e il massimo coefficiente ci per i=1,…,m+K. Grazie al supporto compatto possiamo aggiungere che dato un $x \in [t_l, t_{l+1}]$ allora varrà che $\min\{c_i\} \leq s(x) \leq \max\{c_i\}$ per i=l-m+1,…,l, questa proprietà è detta *supporto compatto*.

Valutazione di una funzione spline

Vediamo adesso come è possibile valutare la nostra funzione spline. Esistono diversi algoritmi utilizzabili, la prima idea è quella di utilizzare direttamente le formule di Cox per valutare le funzioni base non nulle nel punto in cui si valuta la spline e poi applicare la formula per calcolare s(x),

questo approccio non risulta però ottimale. De-Boor propose un algoritmo simile a quello conosciuto per la valutazione delle funzioni di Bezier in cui si risolve la spline tramite l'utilizzo dei soli coefficienti c_i.

Dimostrazione:

Data la funzione spline:

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

Sostituiamo la funzione base $N_{i,m}(x)$ con la sua definizione di ordine inferiore m-1, ottenendo:

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

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

Dato $x \in [t_l, t_{l+1})$, possiamo affermare che le funzioni base di ordine m-1 non nulle per quel punto sono quelle con indice compreso tra l-m+2 e l. Effettuiamo dunque alcune modifiche alle sommatorie in modo da utilizzare le sole b-spline diverse da zero; la prima sommatoria con indice di base i la facciamo variare da i=l-m+2 fino a i=l, mentre per la seconda relativa ad indice di base i+1 impostiamo i=l-m+1,...,l-1. Come ultimo passaggio scorriamo l'intervallo dei valori della sommatoria di +1, sostituendo opportunatamente gli indici, così ottenendo:

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

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

Poniamo:

$$c_i^{[1]} = \left(\frac{c_i^{[0]}(x - t_i) + c_{i-1}{}^{[0]}(t_{i+m-1} - x)}{t_{i+m-1} - t_i} \right) con\ c_i{}^{[0]} = c_i$$

La nostra funzione spline espressa con funzioni base di ordine m-1 diventa:

$$s(x) = \sum_{i=l-m+2}^{l} c_i^{[1]} N_{i,m-1}(x)$$

A questo punto possiamo pensare di iterare il ragionamento per m volte fino ad ottenere che il valore della spline in un punto espresso in termini di funzioni base di ordine 1 è:

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

Il valore della spline in un punto $x \in [t_l, t_{l+1})$ può essere dunque calcolato tramite una corretta combinazione dei coefficienti c_i, iterando per un numero di volte pari a m, lo schema che si crea è il seguente:

$$\begin{array}{cccc} c_{l-m+1}^{[0]} & c_{l-m+2}^{[0]} & \dots & c_l^{[0]} \\ & c_{l-m+2}^{[1]} & \dots & c_l^{[1]} \\ & & \dots & \dots \\ & & & c_l^{[m-1]} \end{array}$$

Espresso anche mediante la formula:

$$c_i^{[j]} = \left(\frac{c_i^{[j-1]}(x - t_i) + c_{i-1}^{[j-1]}(t_{i+m-1} - x)}{t_{i+m-1} - t_i} \right) \text{ per } j=1,\dots,m \text{ e } i=l-m+j+1,\dots,l$$

La complessità computazionale dell'algoritmo è di $O(3m(m-1))$ moltiplicazioni e $O(2m(m-1))$ addizioni.

Costruire una curva B-spline

Tramite i concetti appresi, ci accingiamo adesso a formulare una curva spline, utilizzando la funzione parametrica già vista per le curve spline a nodi semplici.

Dati i vertici di controllo $P_i = \begin{bmatrix} x_i \\ y_i \end{bmatrix}$ (con $i=1,\dots,N$) per poter definire la curva spline è necessario:

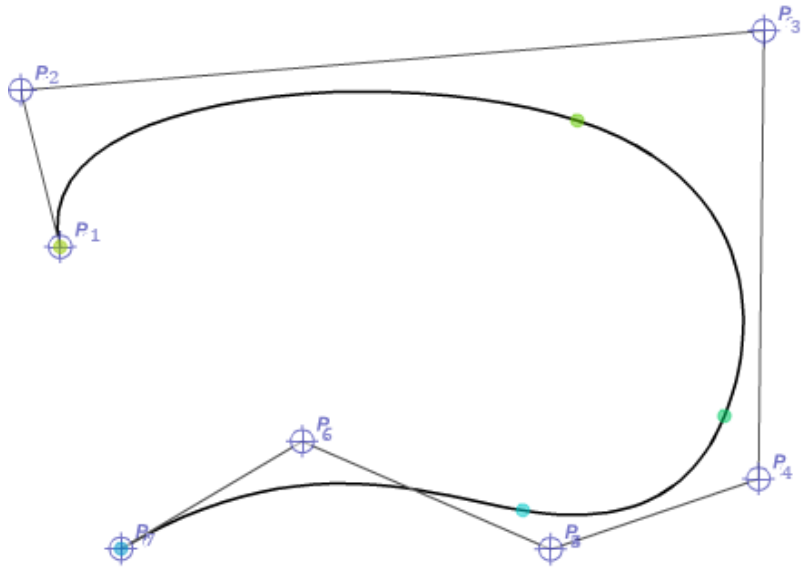
- Fissare l'ordine delle B-spline normalizzate, ovvero m
- Fissare l'intervallo [a,b]
- Fissare numero e molteplicità dei break points

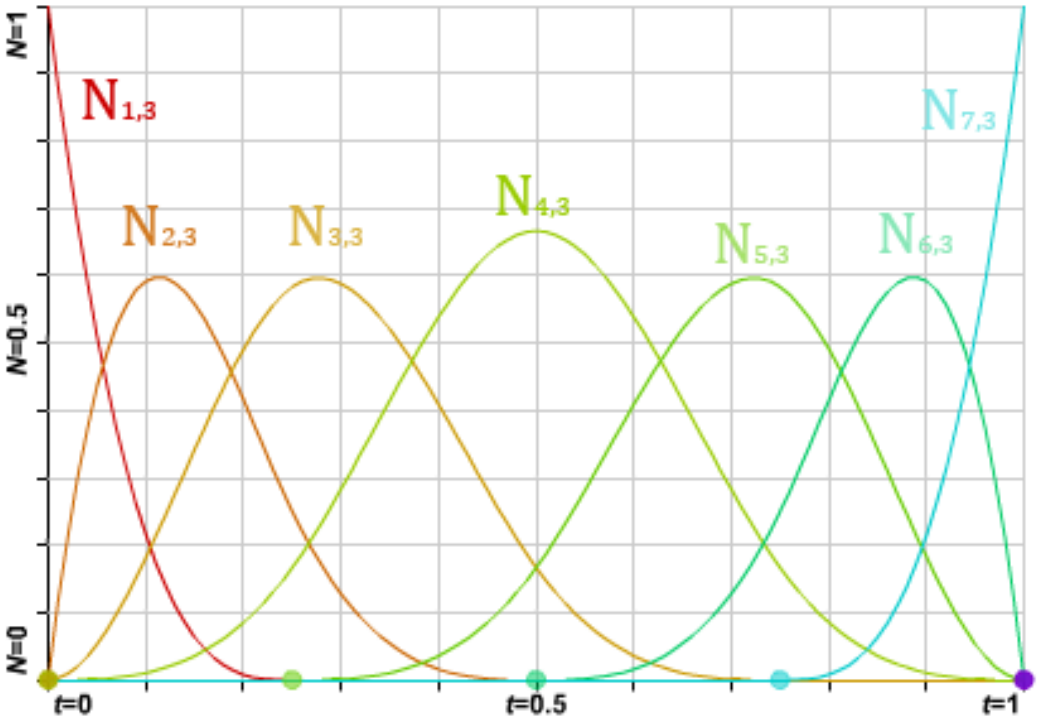
- Definire la partizione nodale estesa, deve valere che il numero dei vertici di controllo N sia pari alla dimensione dello spazio, ovvero N=m+K

Esprimiamo la nostra curva come:

$$C(t) = \sum_{i=1}^{N} P_i N_{i,m}(t) \quad con\ t \in [a, b]$$

Per ricavare la posizione nel piano della curva relativa al parametro t ci basterà a questo punto utilizzare due volte l'algoritmo di De-Boor visto in precedenza, la prima volta con i coefficienti pari a x_i per valutare l'ascissa e la seconda con y_i come coefficienti per valutare l'ordinata.





Nell'esempio vediamo una curva formata da polinomi di terzo grado (m=3) con le rispettive funzioni base b-spline, la partizione nodale estesa è così formata = {0,0,0,0,0.25,0.5,0.75,1,1,1,1}. Si nota come nella partizione nodale estesa i nodi fittizi di destra e sinistra coincidono esattamente con gli estremi dell'intervallo, permettendo a questo punto alla curva di interpolare il primo e l'ultimo punto di controllo.

Proprietà delle curve spline

Soffermiamoci adesso a discutere alcune importanti caratteristiche delle curve spline, le quali ci permetteranno di comprendere al meglio il loro funzionamento. Le proprietà delle curve spline sono una diretta conseguenza delle basi b-spline e da come è formata la partizione su [a,b].

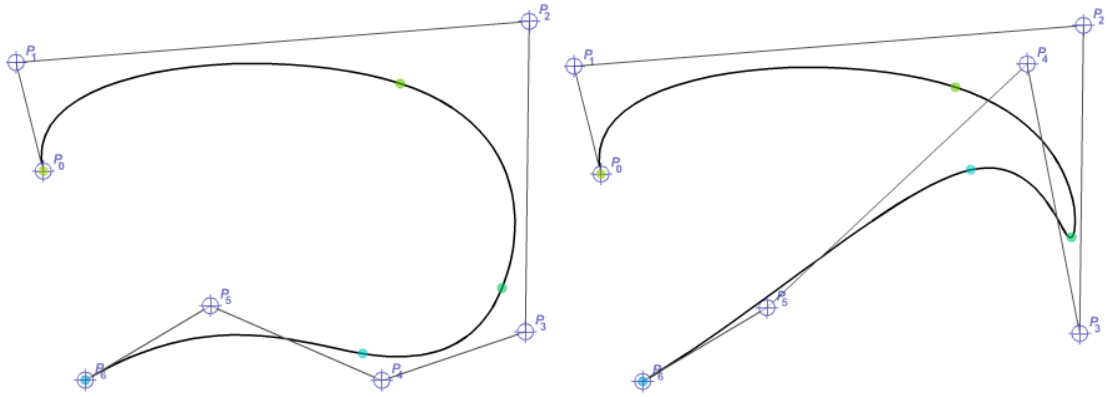
Il numero dei punti di controllo N e l'ordine della spline m è legato dalla relazione:

$$N = m + K$$

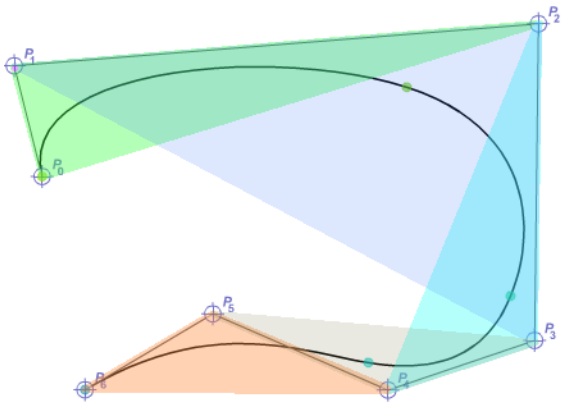
Dove ricordiamo che K è la sommatoria dei valori di molteplicità dei break points nel vettore M, dalla formula si evince il fatto che è possibile utilizzare un numero elevato di punti di controllo pur mantenendo un ordine basso, basterà impostare correttamente i valori di M.

Una curva spline è divisa in spans ovvero parti di curva facenti riferimento a ciascun intervallo internodale, infatti dato un $t \in [t_l, t_{l+1})$ solo i punti Pi con i=l-m+1,…,l influenzano la curva il cui punto è relativo al parametro t (considereazione che deriva ancora dal supporto compatto).

Si può dunque parlare di *controllo locale*, il grande vantaggio delle curve spline a nodi multipli rispetto alle curve di Bezier, per quanto appena detto sappiamo che spostando un punto di controllo P_k la parte di curva influenzata da tale modifica riguarderà solo le spans relative agli intervalli internodali su cui giace la b-spline $N_{k,m}(t)$ (ovvero il supporto della k-esima funzione base).



Un'altra interessante proprietà è quella del forte guscio convesso, ovvero che se $t \in [t_l, t_{l+1})$ allora la curva C(t) è contenuta nell'inviluppo convesso dei punti su cui giacciono le funzioni base non nulle per quel valore di t, ovvero i P_i i=l-m+1,…,l.



La curva B-spline gode della proprietà di *invarianza per trasformazioni affini*, questa risulta essere un grande vantaggio per applicazioni grafiche in cui è spesso necessario modificare la curva, ad esempio durante un animazione. Grazie a questa proprietà possiamo traslare, ruotare e deformare la curva applicando tali trasformazioni solo ai vertici di controllo e poi riapplicare la formula per rivalutare la curva (evitandoci di dover effettuare la trasformazione per ogni singolo valore della curva).

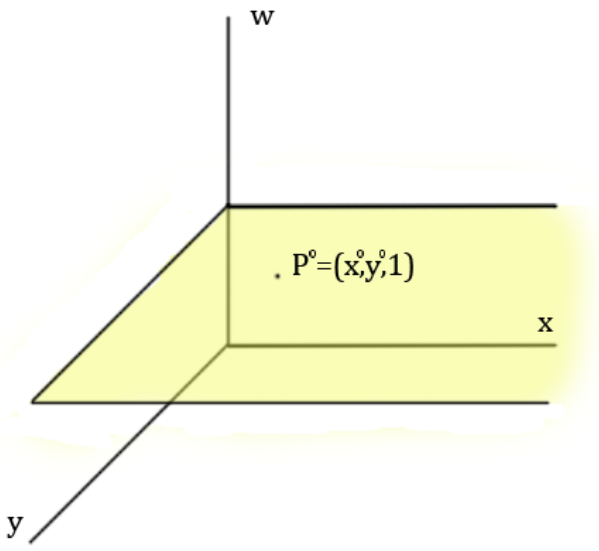
Legame tra curve di Bezier e B-spline

Un particolare da osservare è che le curve di Bezier sono un caso speciale delle curve B-spline. Si consideri una spline di ordine $m=N$, poiché vale la condizione $N=m+K$, si ha che $K=0$, ovvero non si hanno nodi interni. Di conseguenza la partizione nodale estesa è data da N nodi coincidenti con il primo estremo dell'intervallo e N nodi coincidenti con il secondo estremo dell'intervallo. Quindi la curva B-spline di ordine m coincide con la curva di bezier di grado m-1. Come nelle curve di Bezier, lo spostamento di un vertice di controllo influisce tutta la curva.

Curve Spline razionali

Le curve spline viste fino ad ora sono sufficienti per adempiere agli scopi previsti dal progetto, tuttavia queste soffrono ancora di diversi lievi difetti, la cui soluzione sarebbe interessante approfondire. Ricordiamo che le curve B-spline sono curve polinomiali e per questo motivo non sono adatte a rappresentare molte utili forme semplici come cerchi ed ellissi, inoltre non godono della proprietà di invarianza per trasformazioni proiettive, risultando meno pratiche nel passaggio da 3D a 2D. Introduciamo una generalizzazione delle B-spline, chiamata NURBS (Non-Uniform Rational B-Spline).

Possiamo pensare ad una curva razionale come alla proiezione di una curva dello spazio 3d sul piano proiettivo w=1.



Ogni punto $P^0=(x^0,y^0,1)$ può essere considerato come una proiezione di un punto $P=(w^0x^0,w^0y^0,w^0)$ sul piano proiettivo. Consideriamo la nostra curva

spline formata dai punti $P_i=(x_i,y_i)$ i=1,…,N, vogliamo aggiungere alla nostra funzione parametrica una nuova componente $w_i>0$ che vuole riflettere la proiezione precedentemente esposta diventando:

$$C(t) = \begin{cases} \sum_{i=1}^{N} w_i x_i N_{i,m}(t) \\ \sum_{i=1}^{N} w_i y_i N_{i,m}(t) \\ \sum_{i=1}^{N} w_i N_{i,m}(t) \end{cases} => \begin{cases} \frac{\sum_{i=1}^{N} w_i x_i N_{i,m}(t)}{\sum_{i=1}^{N} w_i N_{i,m}(t)} \\ \frac{\sum_{i=1}^{N} w_i y_i N_{i,m}(t)}{\sum_{i=1}^{N} w_i N_{i,m}(t)} \end{cases}$$

Se adesso poniamo:

$$R_{i,m}(t) = \frac{w_i N_{i,m}(t)}{\sum_{i=1}^{N} w_i N_{i,m}(t)}$$

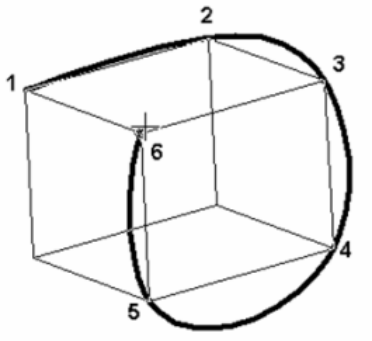
Otteniamo la nostra curve espressa come combinazione delle basi b-spline razionali:

$$C(t) = \sum_{i=i}^{N} P_i R_{i,m}(t)$$

Da notare come se $w_i=1$ per ogni i=1,…,N, ricadiamo nel caso delle curve non razionali, poiché otterremmo che $R_{i,m}(t)$ equivale a $N_{i,m}(t)$ per via della proprietà di partizione dell'unità.

Le NURBS sono l'unica primitiva in matematica che ci permette di descrivere una sezione di una conica e modellare in maniera libera, casi particolari delle NURBS sono le curve B-spline intere, le curve di bezier razionali, le curve di bezier intere.[3] Le NURBS godono di tutte le proprietà delle curve B-spline e in più sono invarianti per trasformazioni proiettive.

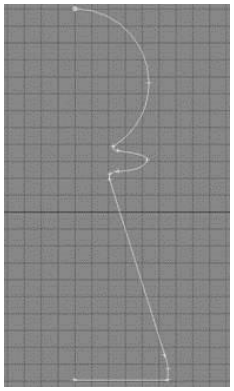
Curve Spline 3D

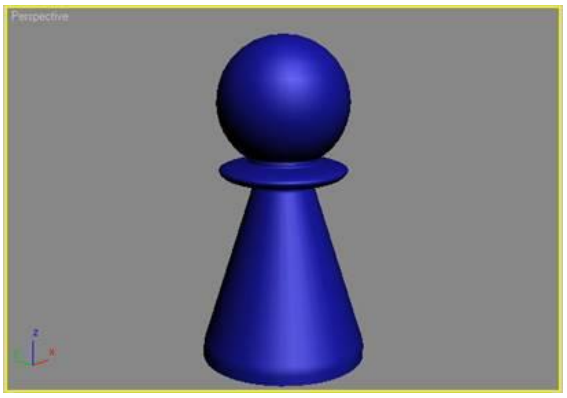


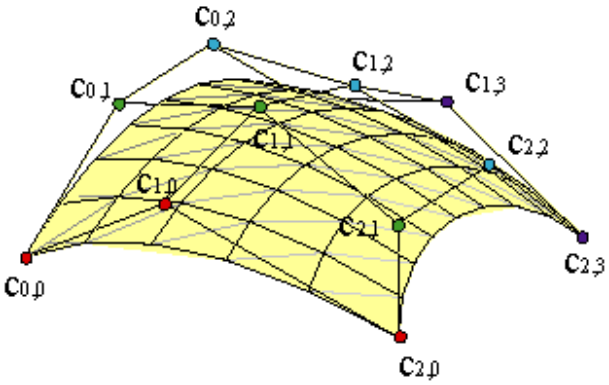
Fino ad ora abbiamo parlato di curve descritte su un piano, tramite i concetti fino ad ora espressi non dovrebbe essere difficile immaginare come poter descrivere una curva spline in uno spazio a tre dimensioni. Ogni punto nello spazio è definito da un vettore di lunghezza tre $P_i = [x_i \quad y_i \quad z_i]$ in cui la nuova componente indica la profondità del punto nello spazio definito dagli assi xyz. Ricordando l'espressione parametrica della curva ci sarà necessario solo aggiungere una nuova funzione spline che valuti i valori di profondità della curva per ogni valore della variabile t, otteniamo:

$$C(t) = \begin{cases} \sum_{i=1}^{N} x_i N_{i,m}(t) \\ \sum_{i=1}^{N} y_i N_{i,m}(t) \\ \sum_{i=1}^{N} z_i N_{i,m}(t) \end{cases}$$

L'applicazione delle spline nell'ambito del 3D si espande anche alla definizione di superfici nello spazio, diverse tecniche sono introdotte per la creazione di forme e modelli, come ad esempio quella per rotazione, dove un oggetto è creato definendo una curva e facendola ruotare introno ad un asse.







Interessante è sapere cosa si può creare combinando due diverse curve, si pensi di voler descrivere una figura nello spazio attraverso l'approssimazione di punti appartenenti ad un control-net. Definiamo una griglia formata da curve orizzontali e verticali che descrivono l'andamento della superfice nello spazio, specifichiamo la nostra figura tramite l'utilizzo di due funzioni spline basate sui parametri indipendenti *t* e *s*.

$$P_{i,j}\ i = 1, \dots, N\ j = 1, \dots, S$$

$$\Delta_1^* = \{x_i\}_1^{m+2K}\ \ \Delta_2^* = \{y_j\}_1^{n+2H}$$

$$S_m(\Delta_1^*, M_1)\ \ S_n(\Delta_2^*, M_2)$$

$$C(t,s) = \sum_{i=1}^{m+K} \sum_{j=1}^{n+H} P_{i,j} N_{i,m}(t) N_{j,n}(s)$$

Quella che otteniamo è una superfice inscritta in un "poliedro di controllo", in cui il valore della forma in un punto (t,s) con t ∈ [x_l,x_{l+1}) e s ∈ [y_f,y_{f+1}) è dettato dalle funzioni base b-spline che cadono in quei precisi intervalli. Le

proprietà delle funzioni spline definiscono delle caratteristiche a questo tipo di superficie, la variazione di un punto del poliedro di controllo comporta una modifica locale della figura, valgono le proprietà di trasformazione affine.[4]

Capitolo 2

Introduzione a Unity3D

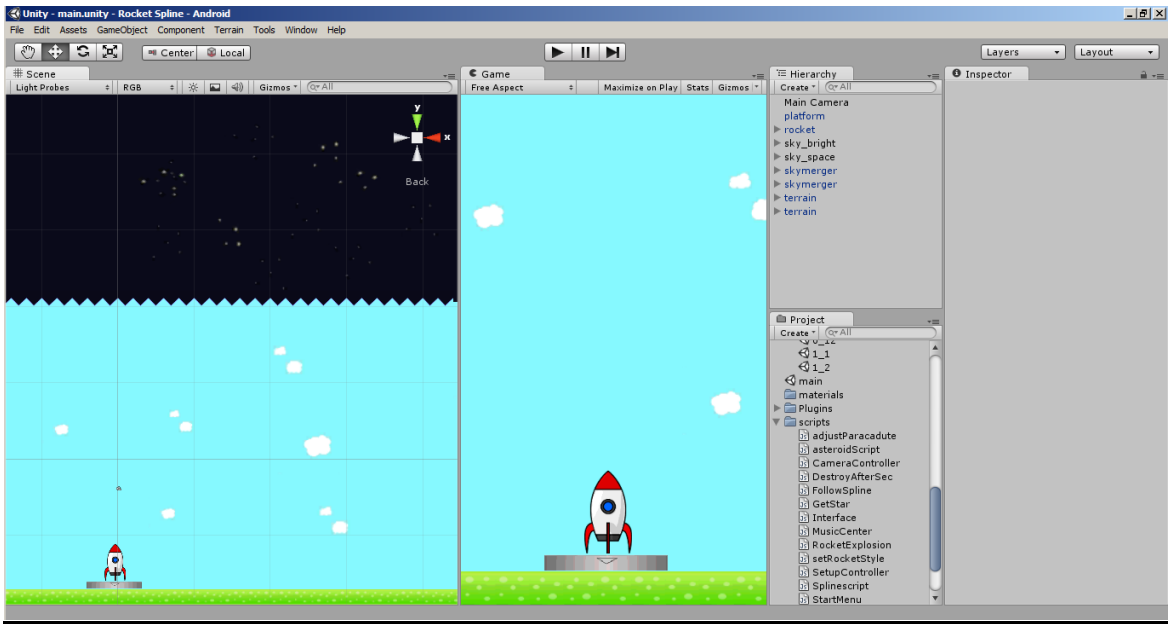
Unity[5] è un ambiente di sviluppo per videogiochi, provvisto di un potente render grafico, che permette un approccio abbastanza semplice a chi si avvicina per la prima volta al mondo della programmazione di videogiochi. In Unity sono presenti diversi tool e oggetti utili nella creazione veloce di contenuti 3D interattivi, quali l'animation tool, il gestore dei materiali o il tool per la creazione di alberi, tuttavia non è presente alcun supporto alla creazione di modelli a tre dimensioni, viene però permesso di importare assets da diversi conosciuti programmi di grafica, come Maya o 3D Studio Max. Grazie al supporto di diverse librerie grafiche quali Direct3D, OpenGL e OpenGL ES Unity permette di costruire applicazioni cross-platform. Tra i diversi sistemi supportati di default sono presenti Windows, Mac, Linux e Web, licenze possono essere acquistate per poter creare giochi per Xbox360, PlayStation3, Wii, Android e iOS.

Workspace

L'area di lavoro di Unity è suddivisa nelle seguenti finestre:

- Scene è la finestra che contiene la vista sulla scena attualmente aperta, in questa finestra è possibile collocare, spostare, scalare e ruotare gli oggetti nello spazio creando l'ambiente di gioco. Nell'angolo in alto a destra è presente un cubo che permette di ruotare la telecamera sulle più comuni viste. È possibile avvicinarsi e allontanarsi dalla scena utilizzando la rotella del mouse, ruotare secondo un punto tenendo premuto il tasto destro del mouse e spostarsi usando il tasto centrale trascinando il mouse.

- Hierarchy è la finestra che elenca tutti gli oggetti presenti nella scena, grazie ad esso è possibile selezionarli e organizzarli. Un oggetto può contenerne altri al suo interno.
- Project è la finestra che ci permette di visualizzare tutte le risorse del nostro progetto (modelli, materiali, audio, script, ecc).
- Game è la finestra che mostra quello che è visibile attualmente dalla main camera, durante la fase di testing il gioco è eseguito al suo interno.
- L'Inspector visualizza le proprietà e i componenti dell'oggetto (game object) selezionato.



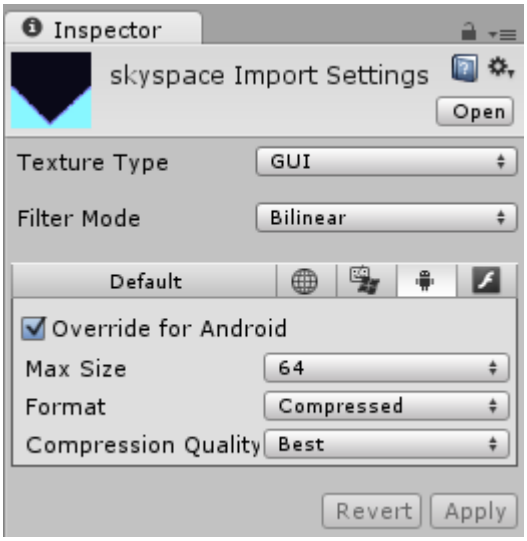
Concetti fondamentali

Ogni progetto creato con Unity deve contenere tutte le risorse di cui l'applicazione ha bisogno, al suo interno possono essere definite nuove scene, scripts e prefabs.

Una scena è un ambiente virtuale isolato, in essa sono definiti tutti gli oggetti che compongono un livello. Implicitamente quando viene caricata la

scena l'engine dealloca tutti gli oggetti in memoria, diverse modalità possono essere utilizzate per mantenere elementi tra una scena e l'altra.
In Unity prevalgono i concetti di *componente* e *game object*. Il game object è la primitiva fondamentale dell'engine a cui si possono aggiungere dei componenti che ne specificano dei comportamenti e delle proprietà. Componente fondamentale di un game object è il Transform, che definisce la posizione, rotazione e grandezza dell'oggetto secondo un sistema di coordinate xyz, tramite un pulsante presente nella parte superiore della finestra principale è possibile variare i valori del componente passando da un sistema di riferimento locale in cui valori sono calcolati in base alla posizione dell'oggetto padre ad un sistema di riferimento del mondo, che si riferisce direttamente allo spazio 3D della scena. Altri componenti importanti comunemente usati sono Camera, Mesh Renderer, Collider, Animation.
Anche gli scripts sono considerati come dei componenti e possono essere aggiunti ad un game object semplicemente trascinando lo script dalla finestra project all'Hierarchy sull'oggetto desiderato. Dall'Inspector è possibile abilitare o disabilitare un componente, come anche settare il valore di alcune variabili.
Unity permette il salvataggio di oggetti creati nella scena tramite l'utilizzo dei *prefab*, elementi creabili dalla finestra project che formano un collegamento immaginario tra tutti i game object dello stesso tipo. Il vantaggio di utilizzare i prefab risiede non solo nel poter salvare all'interno del progetto i game object più comuni, ma anche nel permettere che una modifica effettuata su di esso si ripeta su tutte le copie di quell'oggetto, rendendo molto più facile la gestione delle copie.

Comprimere le risorse



Durante la fase di building Unity comprime le risorse per rendere l'applicazione più leggera. Essendo Unity pensato multi piattaforma è possibile decidere il livello di compressione di file audio, immagini e modelli per ogni diverso sistema. Le risorse sono decompresse all'avvio del gioco e quando necessario anche all'avvio di una scena, a seconda del livello di compressione e del numero di risorse quest'operazione può richiedere più o meno tempo, gli algoritmi utilizzati da Unity sono ottimizzati per rendere l'attesa più breve possibile, per un applicazione mobile solitamente si preferisce privilegiare il peso dell'applicazione.

Per ogni immagine nel progetto esistono due diversi modi in cui l'engine può disegnarla:

- Texture: è il tipo di immagine che si utilizza per rivestire i materiali; il render grafico effettua diverse considerazioni prima di disegnarla, come ad esempio il calcolo della distanza tra l'oggetto e la telecamera per poter sfruttare al meglio le risorse (se l'oggetto è lontano si utilizza meno memoria, diminuisce la qualità). La dimensione dell'immagine deve essere rigorosamente in potenza di due, diversamente Unity la converte aggiungendo pixel extra.
- GUI: solitamente utilizzato nelle interfacce grafiche. Il render grafico la disegna rispettando le dimensioni richieste senza effettuare altre trasformazioni. Questo tipo di immagine non è soggetta al repeat-x o repeat-y.

Nella creazione di un gioco in 2D risulta molto più efficace a volte utilizzare il metodo GUI anche per immagini che non sono utilizzate

nell'interfaccia grafica, in questo modo si salva spazio e si ottiene una migliore qualità dell'immagine.

Dettagli sullo scripting

La parte di scripting di Unity è definita su Mono, un'implementazione open-source del .NET Framework. L'ambiente di sviluppo mette a disposizione dei programmatori diversi linguaggi: UnityScript (un linguaggio quasi proprietario spesso chiamato JavaScript dalla community e dal software), C# e Boo (con sintassi simile a Python). In un progetto si possono utilizzare script scritti in linguaggi diversi, per lo sviluppo di questo progetto si è scelto di utilizzare esclusivamente JavaScript che risulta essere il più comunemente usato dalla community. A partire dalla versione 3.0 l'installazione di Unity include una versione proprietaria di MonoDevelop da poter utilizzare per scrivere codice.

Funzioni base

Come detto in precedenza, ogni script per funzionare necessita di essere "attaccato" ad un game object presente nella scena, diverse funzioni all'interno di uno script sono richiamate solo al verificarsi di un particolare evento relativo a quell'oggetto. Tra le funzioni base più importanti ci sono:

- Update: chiamata prima che un frame proceda nella fase di rendering, al suo interno è spesso specificata la parte di codice che definisce il comportamento dell'oggetto.
- FixedUpdate: come per Update, ma viene chiamata prima di ogni step dell'engine fisico. Utilizzata per oggetti che implementano la fisica di Unity (RigidBody).
- OnGUI: richiamata continuamente per poter disegnare su schermo l'interfaccia grafica e gestire gli eventi ad essa relativi. Perciò

OnGUI potrebbe essere richiamata più di una volta per ogni rendering del frame. Se la proprietà enabled di MonoBehaviour è settata a false, allora la funzione non sarà mai richiamata.

- Start: richiamata quando l'oggetto è caricato nella scena.
- OnTriggerEnter: chiamata quando l'engine effettuando un controllo sulle coordinate si rende conto che due collider si intersecano. Viene chiamata solo una volta.
- OnTriggerExit: come la precedente ma viene chiamata quando i collider non si intersecano più.
- OnTriggerStay: richiamata ogni volta che è verificato che i due collider si intersecano, si ripete ad ogni frame.
- OnCollisionEnter: chiamata quando il collider/rigidbody ha iniziato a toccare un altro collider/rigidbody. (Per il corretto funzionamento è necessario che uno dei due game object abbia il componente Rigidbody).
- OnCollisionExit: come la precedente, chiamata quando il collider/rigidbody ha smesso di toccare un altro collider/rigidbody.
- OnCollisionStay: chiamata ogni frame per il quale il collider/rigidbody sta toccando un altro collider/rigidbody.

Le variabili al di fuori delle funzioni sono inizializzate al caricamento dell'oggetto. Ogni proprietà resa pubblica tramite il prefisso public è resa visibile nell'inspector e permette di poterne assegnare i valori dall'interfaccia di Unity.

Vettori

Fondamentale per ogni buon programmatore di videogiochi è capire come funzionano i vettori. Come abbiamo detto in precedenza ogni game object nella scena presenta di default il componente Transform, al suo interno tre

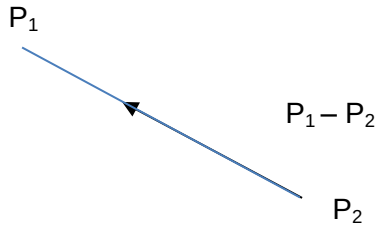
diversi oggetti di tipo Vector3 sono accessibili per poterne definire posizione, dimensione e grandezza nello spazio.

```
function Start () {
    var pos : Vector3 = Vector3(5,5,5);
    pos.x += 15;
    var dim : Vector3 = Vector3(2,2,2);
    var rot : Vector3 = Vector3(90,0,0);
    //assegno i valori al mio gameobject
    transform.position = pos;
    transform.localScale = dim;
    //Passiamo da angoli di Euler a Quaternion
    transform.rotation = Quaternion.FromToRotation(rot, Vector3.up);
}
```

Vector3 è l'oggetto utilizzato per istanziare un nuovo vettore, questi può essere utilizzato non solo per definire una posizione, ma anche uno spostamento, contenendo informazioni sulla direzione, verso e intensità del movimento. Lo stesso discorso si può applicare anche alla rotazione e alla dimensione del game object, diciamo che in generale un vettore può essere utilizzato per creare delle semplici trasformazioni su di essi. Nell'esempio vediamo come è possibile spostare un oggetto della scena in una direzione, ricordandoci di utilizzare la variabile deltaTime della classe Time, che permette di rendere uniforme il movimento per ogni diverso tipo di processore.

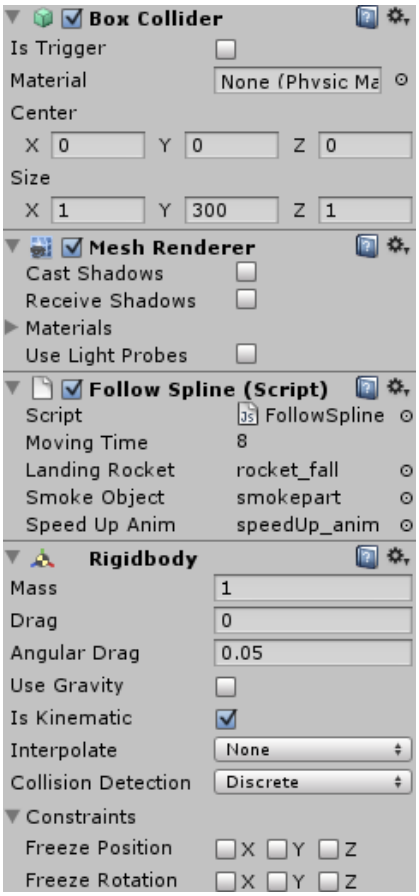
```
function Update () {
    muovi(Vector3.up, 0.5);
}
function muovi(dir : Vector3, speed : float)
{
    //sfrutto solo il verso del vettore
    //e lo moltiplico per una mia intensità
    dir = dir.normalized * speed;
    transform.position += dir * Time.deltaTime;
}
```

Matematicamente è possibile calcolare una direzione dati due punti semplicemente effettuandone la differenza, la direzione di questo vettore è la retta passante per i due punti, il verso è diretto verso il punto sottratto e l'intensità è proporzionale alla distanza tra i due punti.




```
// vettore che punta dal giocatore al target
var heading = target.position - player.position;
```

Componenti comuni



In Unity esistono tanti componenti messi a disposizione dello sviluppatore, molti di questi sono essenziali per la creazione di un gioco, altri servono invece solo a dare particolari effetti grafici. Da codice si possono raggiungere i componenti attraverso la funzione *GetComponent* derivata da *GameObject*. Comune ad ogni componente è la proprietà *enabled*, che permette di attivare/disattivare quel determinato elemento. Diversi components essenziali sono direttamente richiamabili dal gameObject come fossero delle proprietà, ad ed esempio: renderer, audio, animation, collider, rigidbody, particleEmitter, ecc. Vediamo di seguito di spiegare alcuni tra i componenti essenziali:

- Transform: l'abbiamo già trattato, si

occupa di mantenere informazioni su posizione, rotazione e dimensioni del game object. La sua classe è dotata di funzioni che permettono di effettuare delle semplici trasformazioni sugli oggetti (Translate, Rotate, LookAt, ecc). TransformDirection e TransformPoint sono funzioni che permettono di convertire una direzione o un punto da coordinate locali a globali, viceversa per InverseTransformDirection e InverseTransformPoint.

- Renderer: è il componente che si occupa di disegnare a video l'oggetto, disattivando il renderer rendiamo invisibile il game object. Tramite questo componente possiamo accedere ai materiali della mesh e ad esempio cambiarne le texture via codice.
- AudioSource: è la sorgente del suono attaccata ad un game object. Diversi sono i parametri settabili, le funzioni principali per gestire i suoni sono Play e Stop. L'audio source di Unity di default emula l'effetto doppler.
- Camera: crea una finestra sulla scena a partire dalla posizione di quel game object. Si può settare sia la vista ortografica che prospettica, diverse telecamere possono essere presenti nella scena, ma solo una di esse può essere attiva per volta (è comunque possibile implementare lo split screen). La classe Camera mette a disposizione diverse funzioni utili per poter convertire un punto da spazio a schermo e viceversa.
- Collider: attiva il collision detection per quell'oggetto, permette di poter utilizzare funzioni quali OnCollision enter. Ad ogni collider è associata una mesh, unity di default utilizza un cubo, ma si possono anche utilizzare sfere e altre forme geometriche. Settando il parametro isTrigger renderemo il collider un "interruttore", che non deve più occuparsi di fare calcoli fisici per gli urti diventando gestibile esclusivamente attraverso le funzioni di tipo OnTrigger.

Quando isTrigger non è attivo è fondamentale ricordarsi di aggiungere un rigidbody all'oggetto, soprattutto se si vogliono utilizzare le funzioni che iniziano per OnCollision.

- RigidBody: questo componente assegna il game object all'engine fisico di Unity, aspetti quali gravità, accellerazione e attrito saranno simulati per questo oggetto. I rigidbody sono completamente personalizzabili, è infatti possibile definire centro di massa, velocità angolare, modalità di collisione, massa, ecc. Tramite la proprietà *isKinematic* possiamo decidere quando l'oggetto è realmente affetto dalla fisica, è interessante sapere che le collisioni continuano ad essere calcolate anche quando questa proprietà è settata a true.

Una lista completa dei components con le loro funzioni e proprietà è disponibile presso la Unity Scripting Reference.[6]

Comunicazione tra script

Un qualsiasi script "attaccato" ad un oggetto nella scena, viene trattato come un componente, quando si crea un nuovo script questo diventa immediatamente un nuovo "tipo" identificato dal nome dato allo stesso. Possiamo accedere a funzioni e proprietà pubbliche del component di un determinato oggetto semplicemente richiamando *GetComponent*. Per evitare di passare dal componente è possibile utilizzare la funzione *SendMessage* derivabile dal gameobject, questa si occupa di cercare ed eseguire la funzione il cui nome gli viene passato in input (appartenente a qualsiasi script attaccato a quell'oggetto). Nell'esempio vediamo come è possibile trovare da codice un oggetto nella scena e richiamarne a seguito una sua funzione.

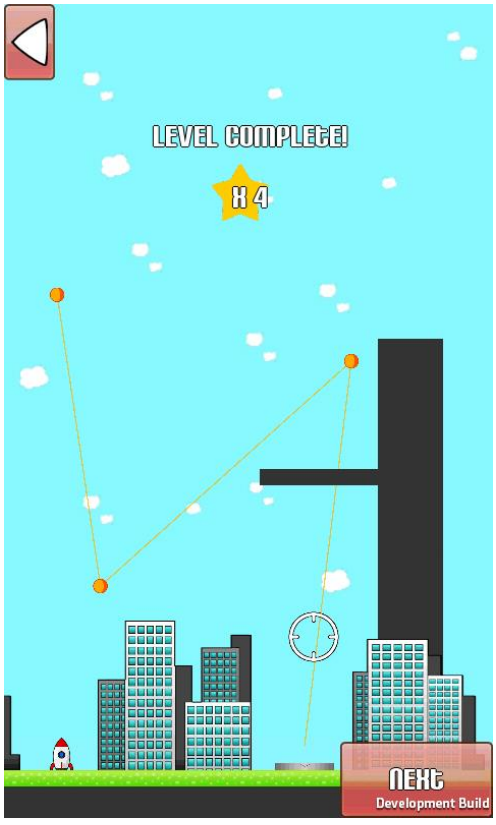
```
function Start()
{
    var controlCenter : GameObject; //dichiaro una variabile di tipo
GameObject
    controlCenter = GameObject.Find("ControlCenter"); //restituisce
l'oggetto ControlCenter della scena
    controlCenter.GetComponent(MusicCenter).toogle_music = true; //modifico
una proprietà dello script

    controlCenter.SendMessage("set_music", true);
//richiamo la funzione set_music da qualsiasi script del controlCenter
    //set_music prende in input un booleano, lo passo come secondo parametro
di SendMessage
}
```

Tramite le funzioni Find e FindGameObjectsWithTag della classe GameObject è possibile ottenere i riferimenti agli oggetti presenti nella scena. Ogni proprietà a cui accediamo tramite GetComponent deve essere *public* o *internal*.

Capitolo 3

Sviluppo dell'idea

L'idea di Rocket Spline è nata a seguito dello studio dei metodi numerici per lo sviluppo di una curva su un piano. Apprese tali conoscenze si è pensato a come poter unire questo concetto al mondo del game design. L'implementazione delle curve risulta avere molte applicazioni nel mondo della creazione di videogame, ad esempio si possono utilizzare per definire delle mesh dinamiche, oppure per creare degli effetti grafici con le particles. Nel progetto si è voluto dare un ruolo più importante alle spline, da questo è nata l'idea di creare una meccanica di gioco che le metta in chiara evidenza.



Si è così sviluppato un modello generale in cui un oggetto che si muove attraverso una curva descritta da punti di controllo di numero predefinito ha l'obiettivo di raccogliere degli oggetti nello spazio evitando degli ostacoli, una meccanica adatta ad un gioco per piattaforma mobile, in grado di intrattenere il giocatore in una serie di livelli sempre più complessi, ricadendo nella categoria di puzzle game.

L'idea del progetto è semplice, per cui non è servito soffermarsi a pensare ad una storia o bonus aggiuntivi, tuttavia la stessa idea di base è stata utilizzata per creare due diverse modalità di gioco, in modo tale

da rendere il tutto più vario. Una volta definito il progetto, si è passati a pensare a quale possa essere un setting per il gioco, ovvero specificare chi sia il giocatore, l'ambiente in cui risiede e quali obiettivi deve compiere. A tale proposito le scelte sono innumerevoli: si è optato per qualcosa che possa risultare facile da disegnare e che permetta in seguito di avere uno stile grafico che si adatti alle capacità dello sviluppatore. Nel lavoro finale vediamo un razzo su una pianura o nello spazio, il tutto in una grafica molto stilizzata.

Prima di iniziare effettivamente la fase di implementazione è indispensabile conoscere e scegliere i tool da utilizzare durante tutto il progetto, Unity risulta essere un ambiente di sviluppo efficace per i motivi visti in precedenza, infatti la possibilità di poter compilare per diverse piattaforme e l'essere gratuito lo rende unico nel suo genere.

Quanto abbiamo appena descritto è un processo di sviluppo dell'idea veloce e adatto a chi crea un videogame semplice a livello concettuale e senza l'aiuto di terze parti, diverse sono le domande che uno sviluppatore si deve porre all'inizio di questa fase, la più importante è forse quella riguardante le capacità del team di sviluppo (o in questo caso del singolo), cercando di mantenere un livello qualitativo che non sfoci nell'impossibile da realizzare, ponendo il lavoro in una situazione di stasi.

Specifiche del progetto

Rocket Spline nasce con l'idea di essere un gioco per piattaforma mobile e PC che implementi come tecnica di gameplay il meccanismo delle curve spline. Il giocatore si vedrà alla presa di diversi livelli che gli richiedono di affrontare dei rompicapo sempre più complessi in cui dovrà costruire una curva attraverso un numero limitato di vertici di controllo. Visibile a

schermo non sarà la curva, ma bensì il suo poligono di controllo, la forma creata sarà utilizzata come percorso di un razzo, il quale dovrà evitare ostacoli e raccogliere oggetti. Lo scopo dei livelli varia a seconda della modalità di gioco. Inizialmente l'utente potrà giocare soltanto alla "*modalità da terra*" il cui obiettivo è raccogliere tutte le stelle nella scena e non far esplodere il razzo, successivamente sbloccherà la "*modalità spazio*" in cui si dovranno far esplodere degli asteroidi mantenendo una velocità elevata. Un livello si può pensare diviso in due diverse fasi di gioco:

Setup Mode: in questa prima fase il giocatore inserirà nella scena i vertici del poligono di controllo, formando mentalmente la curva che descriverà il percorso seguito dal razzo. Il numero disponibile di punti varierà da livello a livello, le caratteristiche della curva resteranno invece invariate. Il primo vertice di controllo è sempre settato alla posizione iniziale del razzo, la curva interpolerà sempre il primo e l'ultimo vertice di controllo. Una volta settati tutti i punti del poligono sarà permesso proseguire nella fase di lancio attraverso l'utilizzo di un pulsante in sovraimpressione. È possibile muovere e ingrandire sulla scena attraverso dei controlli predefiniti per ogni diverso tipo di piattaforma.

Controlli	PC	Mobile
Inserimento punto	Tasto sinistro del mouse	Longpress/GUI
Rimozione punto	GUI	GUI
Modifica punto	Selezione di un punto e selezione sulla nuova posizione	
Movimento camera	Tasto dx e movimento del mouse	Slide
Zoom camera	Rotella del mouse	Pinch to zoom

Launch mode: una volta fatto partire il razzo questi seguirà il percorso specificato attraverso i punti inseriti dall'utente. Ad ogni ostacolo incontrato il razzo esploderà, lasciando al giocatore la

possibilità di ritentare, modificando la posizione dei punti di controllo. Durante questa fase saranno presenti due diversi pulsanti a schermo a seconda della modalità di gioco, nella *land mode* il giocatore dovrà premere nel momento più adatto il pulsante RETRIEVE, il quale permette l'atterraggio del razzo, facendo attenzione a farlo cadere su una "zona sicura". Nella *space mode* invece sarà presente il pulsante BOOST, che aumentando la velocità del razzo faciliterà la distruzione delle asteroidi.

Si è deciso di investire nell'implementazione di due diverse modalità di gioco perché questi non risultasse alla lunga troppo ripetitivo, pur essendo un puzzle game. Le differenze tra le due modalità, a parte la grafica, sono minime, ma sostanziali. In *land mode* il giocatore farà più attenzione a come meglio costruire la curva, cercando di diventare sempre più preciso nella raccolta delle stelle, imparerà come verrà sviluppato il percorso in base al poligono di controllo, diversamente in *space mode* questi dovrà tener conto del variare della velocità del razzo nel tempo, imparando che la velocità del razzo in un punto della curva dipende dalla distanza tra i vertici di controllo.

Le scene

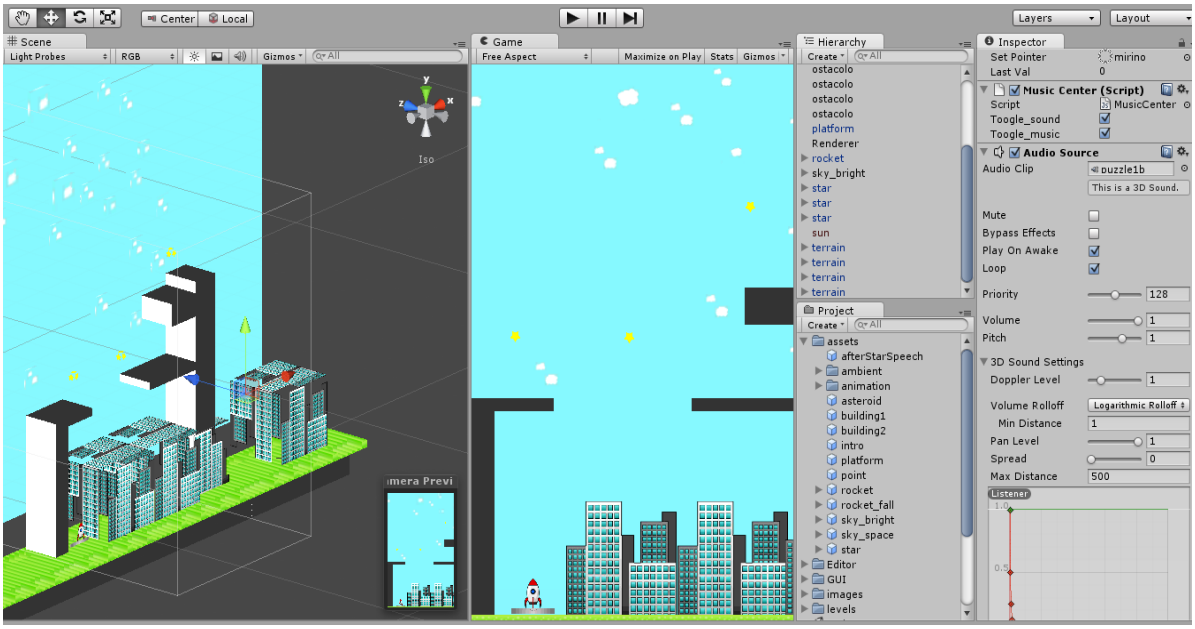
Per il progetto è stato scelto di usare una grafica 2D. Unity nasce come ambiente di sviluppo per videogame a tre dimensioni, ma attraverso dei materiali adatti ed una telecamera ortografica si possono ottenere degli ottimi effetti 2D: basta restringere il tutto su due piani (si esclude la profondità z). Ogni livello è salvato in una scena diversa, molti degli oggetti presenti sono dei prefab, rendendo la gestione degli elementi meno complessa. I principali oggetti ripetuti nella scena sono i seguenti:

- Rocket: è il razzo che seguirà la curva, ad esso è associato lo script *FollowSpline* (di cui si vedrà in seguito l'implementazione) che si

occupa di far seguire all'oggetto il percorso creato dai vertici di controllo.

- ControlCenter: è un punto di riferimento per gli altri script, è un empty game object con associato lo script *SetupController* e *SplineScript* che si occupano dei gestire inserimento dei vertici e calcolo della spline.
- Main Camera: è l'oggetto a cui sono assegnati i componenti *Camera*, *AudioSource*, *AudioListener*. Diversi Script sono assegnati necessari per il controllo della telecamera (zoom, movimento) e dell'audio.
- Borders: è un oggetto vuoto i cui figli sono dei collider che definiscono i bordi del livello.

Una volta implementata la logica di un livello questa può essere facilmente ripetuta di scena in scena importando gli stessi oggetti, creare un buon puzzle dipenderà soltanto dalla disposizione di ostacoli e obiettivi (stelle/asteroidi).



Implementazione delle Spline in Unity

Unity non dispone un tool che permette di definire curve, tuttavia sullo store di Unity Technologies è possibile acquistare diverse soluzioni sviluppate dagli utenti.
Vediamo ora un implementazione dell'algoritmo di De-Boor per la valutazione di una curva spline.
Iniziamo con il definire le nostre variabili:

```
// Definizioni variabili globali
public var m : int = 4; //ordine delle funzioni base b-spline che formano la curva, nei livelli con 4 nodi si cade nel caso delle curve di Bezier, poiché K=0

// Definizioni variabili locali
private var c : Vector3[]; //coordinate dei punti che formano il poligono di controllo
private var t : float[]; //variabile t della curva parametrica, valori ottenuti con parametrizzazione
private var N : int; //numero dei punti
private var K : int; //sommatoria dei pesi sui nodi
private var nodi : float[]; //nodi su cui sono definite le b-spline
```

L'ordine della curva è variabile dall'hierarchy, ma si è scelto comunque di mantenere sempre il valore di default 4, che risulta essere ottimo per il progetto, considerato il basso numero di punti da inserire in un livello. La variabile t è un array contenente dei valori compresi tra 0 e 1 calcolati progressivamente tramite tecniche di parametrizzazione, è indispensabile

quando si vuole disegnare la curva. La variabile *c* è un array di *Vector3*, per quanto la terza componente dello spazio risulti ridondante, poiché simulando il 2D tutti i punti avranno la stessa profondità, si è deciso comunque di utilizzare tale tipo di variabile essendo la primitiva che si occupa di definire la posizione di un punto nella scena.
Prima di passare all'algoritmo di De-Boor dobbiamo formare la nostra partizione nodale estesa, la curva che vogliamo ottenere deve interpolare il primo e l'ultimo vertice di controllo, quindi è necessario impostare il massimo grado di molteplicità per tali punti. Per i nodi interni invece si è scelto di mantenere molteplicità pari a uno.

```
function assegnaNodi()
{
    //Nel vettore avrà partizione dei nodi con ripetizioni così come segue
    //impongo il passaggio della curva per il primo vertice di controllo
    for(var i = 1; i<=m; i++)
        nodi[i] = 0; //min t

    //imposto molteplicità ai nodi centrali ( m+1 -> m+k )
    //escludo dalla scelta dei nodi i parametri t2 e tk-1 (uno prima
degli estremi)
    var j = 3;
    for(i = m+1; i<=m+K;i++){
        nodi[i] = t[j];
        j++;
    }
    //nodi fittizzi a destra
    for(i = m+K+1; i<=K+2*m; i++)
        nodi[i] = 1; //max t
}
```

Se si vuole disegnare la curva, è indispensabile implementare la parametrizzazione, la quale ci permette di mantenere i valori della variabile parametrica per i quali vogliamo conoscere la posizione dello spazio nella curva. Considerando t inizializzato come un vettore di lunghezza pari a 1000, vediamo come implementare la parametrizzazione uniforme e delle corde:

```
//Funzioni di parametrizzazione
//la parametrizzazione è sempre basata su un intervallo [0,1]

function parametrizzazioneUniforme(){
    var i : int;
    var step : float;
    t[0] = 0;
    step = 1.0/(N-1);
    for(i=1;i<=N;i++)
        t[i]=t[i-1]+step;
}

function parametrizzazioneCorde(){
    var i : int;
    t[1] = 0;
    for(i=2;i<=N;i++){
        //Debug.Log(i);
        t[i]=t[i-1]+Mathf.Sqrt((c[i-1].x - c[i-2].x)*(c[i-1].x - c[i-
2].x)+(c[i-1].y - c[i-2].y)*(c[i-1].y - c[i-2].y));
        }
    //Mappo t in [0,1] per normalizzare il parametro
    for(i=2;i<=N;i++)
        t[i]=t[i]/t[N];
}
```

Necessario per l'algoritmo di De-Boor è conoscere in quale intervallo internodale cade il valore della variabile parametrica, implementiamo una funzione che tramite la bisezione ci restituisca l'indice di tale intervallo:

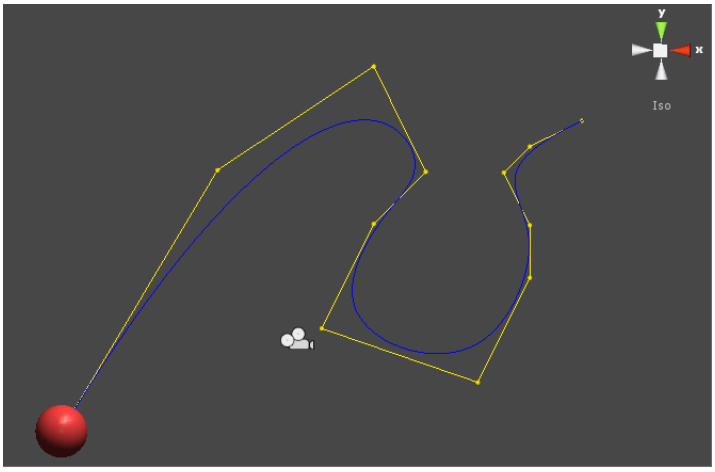
```
function bisezione(z : float)
{
    /*
    Funzione di bisezione per localizzare in quale intervallo internodale
    [tl,tl+1) appartiene il valore z
    */
    var l = m;
    var u = m+K+1;
    while((u-l)>1){
          var mid = (l+u)/2;
          if(z < nodi[mid])
             u = mid;
          else
             l = mid;
    }
    return l;
}
```

A questo punto non ci resta che implementare l'algoritmo per la valutazione della spline. Creiamo una funzione che preso in input il valore della variabile parametrica $tg \in [0,1]$ ne restituisce il punto nello spazio della curva definita tramite i vertici specificati nel vettore c per quel dato valore.

```
function valutaSpline(tg : float){
    //Valuta la curva spline in un valore del parametro t dato tg
    //il punto tg è compreso nell'intervallo [0,1]
```

```
    var l : int;
    var cx : float[] = new float[m+1]; //ricorsione dei punti di
controllo per x
    var cy : float[] = new float[m+1]; //ricorsione dei punti di
controllo per y

    //localizzo l'intervallo internodale in cui cade tg
[nodi(l),nodi(l+1)]
    l = bisezione(tg);

    //al primo passo copio i coeff
    for(var j = 1; j<=m; j++)
    {
        cx[j] = c[j+l-m-1].x;
        cy[j] = c[j+l-m-1].y;
    }

    //eseguo l'algoritmo di De Boor
    for(j = 2; j<=m; j++)
    {
        for(var i = m; i>= j; i--)
        {
            var denom : float = nodi[i+l-j+1]-nodi[i+l-m];
            var a1 : float= (tg-nodi[i+l-m])/denom; // primo coeff
            var a2 : float = 1-a1; //secondo coeff
            cx[i] = a1*cx[i]+a2*cx[i-1];
            cy[i] = a1*cy[i]+a2*cy[i-1];
        }
    }

    var res = new Vector3(cx[m],cy[m],0);
    return res;
}
```

Una volta compreso il funzionamento dell'algoritmo di De-Boor, la sua implementazione risulta abbastanza intuitiva. Come primo passo si individua in quale intervallo internodale risiede il punto della variabile parametrica tg presa in input, successivamente si definiscono due array di lunghezza *m+1* su cui andremo ad iterare l'algoritmo, calcolando di volta in volta i valori dell'ascissa e dell'ordinata di c relativi ad uno specifico passo, fino ad ottenere il valore relativo al punto *tg*.



Prima di richiamare la funzione valutaSpline è importante inizializzare i valori della curva quali la partizione nodale estesa o il vettore t, perciò si utilizza la seguente funzione che presi in input i vertici di controllo nello spazio si occupa di settare le variabili necessarie per poter utilizzare valutaSpline:

```
function instantiateSpline (c_points : Vector3[]) {

    c = c_points;
    N = c.Length;
    t = new float[1000];
    K = N - m;
    nodi = new float[2*m + K + 1];


    if(usaParamCorde)
        parametrizzazioneCorde();
```

```
    else
        parametrizzazioneUniforme();

    assegnaNodi();
    init_complete = true;
}
```

Muovere un oggetto lungo la curva

Abbiamo visto come tramite la funzione valutaSpline possiamo ottenere la posizione della curva nello spazio relativa ad un preciso valore della variabile parametrica, sapendo che questa varia tra [0,1] ci serve solo un metodo che progressivamente nel tempo la incrementi. Per fare ciò possiamo utilizzare la proprietà *deltaTime* della classe *Time* che restituisce il lasso di tempo trascorso tra il rendering del corrente frame con il precedente, ottenendo anche un modo stabile per rendere uniforme il movimento su processori differenti.

```
private var t : float = 0;
private var newPos : Vector3;

function Start () {
    splineController =
GameObject.Find("ControlCenter").GetComponent(Splinescript);
    //ottengo lo script che mi valuta la spline, precedentemente
assegnato all'oggetto ControlCenter
}

function Update () {

    var inc : float = (Time.deltaTime/movingTime) * speed;
```

```
    t += Mathf.Clamp(inc,0,0.0035); //limito la velocità del razzo ad un massimo incremento
    newPos = splineController.valutaSpline(t);
    transform.LookAt(newPos, Vector3(0,0,1)); //ruota sull'asse z locale
    transform.position = newPos;

}
```

Tramite valutaSpline otteniamo la posizione che l'oggetto deve ottenere nello spazio ad ogni chiamata di *Update()*, con movingTime definiamo in secondi il tempo che vogliamo che l'oggetto impieghi nel percorrere la curva, speed invece è un moltiplicatore utilizzato per aumentare la velocità del razzo nella space mode. Nel codice gestiamo anche la rotazione dell'oggetto, volendo che questi si rivolga sempre in modo che il suo asse y sia tangente alla curva nel punto percorso. Per ottenere la nuova rotazione dell'oggetto l'approccio utilizzato non è quello di calcolare la derivata della curva per ogni punto, ma quello di utilizzare la funzione *LookAt* di transform, rivolgendo l'oggetto verso la nuova posizione (motivo per cui utilizziamo una variabile d'appoggio *newPos*). Da notare che non ci sono vincoli nell'incremento della variabile t, permettendo che questa superi il valore massimo di 1, questo perché la curva diventa una retta che continua a mantenere la pendenza che la spline assume nell'ultimo punto con t=1, allungando il percorso dell'oggetto. Quanto appena visto è presente nello script FollowSpline associato all'oggetto *rocket* della scena.

Creare le scene

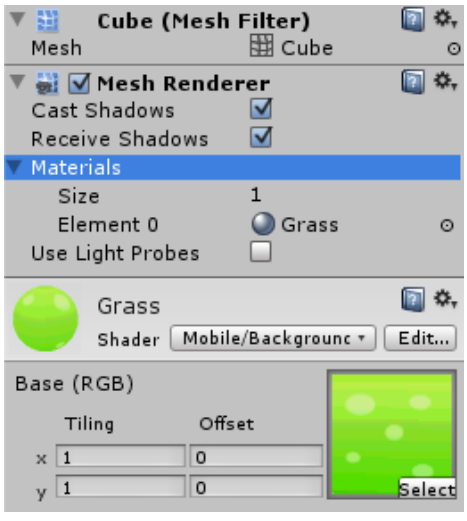
Come abbiamo già detto in precedenza, una scena è un ambiente virtuale isolato in cui è possibile definire un nuovo livello. Possiamo gestire le scene direttamente dalla finestra Project, per creare una nuova finestra

selezioniamo *File > New Scene* dal menù principale. Ogni scena che vogliamo sia inserita nel nostro lavoro deve essere inserita nella lista delle scene da compilare (*Scenes in Build*), per fare ciò selezionaiamo dal menù *File > Build Settings* e premiamo il pulsante *Add Current* per inserire la scena corrente nella lista.

Grafica 2D

Le scene di Unity sono un ambiente a tre dimensioni, non c'è possibilità di cambiare questo aspetto, l'unico modo per poter disegnare a schermo delle primitive a due dimensioni è attraverso la funzione OnGUI via codice, per questo motivo c'è bisogno di simulare il 2D direttamente nella scena, utilizzando qualche accorgimento per migliorarne la qualità grafica.

- Settare la telecamera come ortografica (dal componente selezionare *Projection > Orthographic*), questo tipo di vista permette di annullare l'effetto profondità.
- Fissare una profondità standard da mantenere per tutti gli oggetti che devono interagire sulla scena (ad esempio z=0).
- Per i materiali utilizzare immagini di tipo GUI invece che Texture.



Una grafica 2D solitamente risulta essere semplice e pulita, non sono necessari shader che tengano conto di effetti di luce e ombre, quando si crea un nuovo oggetto a due dimensioni si opta solitamente per i materiali presenti nella categoria Unlit di Unity appartenenti agli Standard Assets importabili nel progetto dal menù *Assets > Import Package*.

Per creare un nuovo oggetto dunque selezioniamo dal menù *GameObject > Create Other > Cube* per creare un

cubo con un materiale di default assegnato, dalla finestra Project creiamo un nuovo materiale di tipo Unlit/Transparent o Unlit/Texture e assegniamo nel campo indicato l'immagine che vogliamo utilizzare per quell'oggetto, infine selezionato il cubo dall'inspector sotto la voce Mesh Renderer sostituiamo il materiale di default con quello appena creato. Se l'immagine appare sproporzionata è necessario modificare altezza e lunghezza del cubo in modo che siano valori proporzionali alla dimensione dell'immagine.

Aggiunta dei componenti

Per dare vita alla scena è necessario assegnare ai game object componenti e scripts, in primo luogo è necessario fornire un collider ad ogni oggetto che nella scena è "solido" ovvero forma un ostacolo nel movimento, diversamente ogni oggetto si attraverserebbe senza alcuna nota l'uno dell'altro. Per aggiungere un componente ad un oggetto selezionato è necessario selezionare dal menù la voce *Component* e navigare nel menù a tendina fino a raggiungere il componente desiderato.

Interfaccia e Input

Fondamenti sulle GUI

In Rocket Spline esistono due tipi di interfacce, la principale è quella che si presenta all'avvio del gioco e che presenta la selezione dei livelli, la seconda comune ad ogni livello è quella utilizzata per compiere delle azioni nel gioco.

Unity permette di definire la grafica di uno specifico elemento della GUI attraverso degli oggetti chiamati *GUIStyle*, definibili via codice e attraverso l'inspector. Proprietà di questi oggetti sono le immagini assegnate per ogni stato dell'elemento (Normal, Hover, Active), font utilizzato dal testo, bordi, margini, padding, ecc.

Tutto quello che deve essere visualizzato nella Graphical User Interface deve essere definito via codice all'interno della funzione OnGUI di cui abbiamo parlato in precedenza; anche gli eventi come la pressione di un bottone sono richiamati a partire da OnGUI.

Posizione e dimensione degli elementi sono gestiti attraverso un tipo di variabile chiamato *Rect*; i valori utilizzati sono relativi alla finestra dell'applicazione e hanno origine nel suo angolo in alto a sinistra. Non è raro che un applicazione giri su schermi che hanno diverse risoluzioni, soprattutto nel caso di dispositivi android, in cui ogni telefono ha uno schermo di dimensioni differenti dagli altri. Il risultato ottenuto con le Rect è quello di elementi di dimensione o posizione differenti a seconda della risoluzione degli schermi.

Per mantenere la GUI invariata è possibile giocare sulle proporzioni, partendo da due valori di altezza e larghezza fissati. Nell'esempio vediamo come applicare una trasformazione sulla matrice della GUI, in modo da mantenerla invariata per ogni diverso tipo di risoluzione:

```
private var originalWidth = 480.0;  // risoluzione scelta per creare
private var originalHeight = 800.0; // i contenuti della GUI
private var scala : Vector3;

function OnGUI()
{
    scala.x = Screen.width/originalWidth; // calcola il rapporto
orizzontale
    scala.y = Screen.height/originalHeight; // calcola il rapporto
verticale
    scala.z = 1;
    var svMat = GUI.matrix;
    // Sostituisco la matrice, solo le proporzioni sono invariate
rispetto l'originale
    GUI.matrix = Matrix4x4.TRS(Vector3.zero, Quaternion.identity, scala);
```

```
    ////////////////////////////////////////////
    //Definizione della GUI
    ////////////////////////////////////////////
    GUI.matrix = svMat; // ripristino la matrice originale
}
```

Una volta definita la matrice di scala possiamo iniziare a definire gli elementi appartenenti alla nostra GUI.
I principali utilizzati nel progetto sono GUI.Label, GUI.Button e GUI.Box. Poiché tutta l'interfaccia è definita in un'unica funzione, si è pensato di utilizzare una variabile state che potesse definire in quale stato dell'interfaccia ci troviamo e disegnare a video gli elementi relativi di conseguenza. Vediamo adesso un esempio di schermata con tre differenti bottoni:

```
private var state : int = 0;
// 0 = house logo
// 1 = start & game logo
// 2 = level select (land)
// 3 = level select (space)
// 4 = ship select
public var optionStyle : GUIStyle;
public var labelStyle : GUIStyle;
```

```
...
//START MENU
    if(state == 1)
    {
        logo_pos_x = originalWidth/2 - gameLogo.width/2;
        logo_pos_y = 5;
        var button_pos_x = originalWidth/2 - 75;
        var button_pos_y = originalHeight/2 + 50;
```

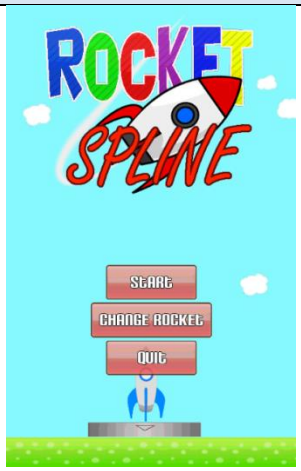
```
        GUI.DrawTexture(Rect(logo_pos_x, logo_pos_y, gameLogo.width,
gameLogo.height), gameLogo);

        if(GUI.Button(Rect(button_pos_x, button_pos_y, 150, 60), "Start",
optionStyle)){
            state = 2;
            if(toogle_sound)
audio.PlayClipAtPoint(clickAudio,transform.position, 1);
            launchRocket();
            }
        DrawOutline(Rect(button_pos_x, button_pos_y, 150, 60), "Start",
labelStyle, Color.black, Color.white);

        if(GUI.Button(Rect(button_pos_x-25, button_pos_y+65, 200, 60),
"Change Rocket", optionStyle)){
            state = 4;
            if(toogle_sound)
audio.PlayClipAtPoint(clickAudio,transform.position, 1);
            }
        DrawOutline(Rect(button_pos_x-25, button_pos_y+65, 200, 60),
"Change Rocket", labelStyle, Color.black, Color.white);

        if(GUI.Button(Rect(button_pos_x, button_pos_y + 130, 150, 60),
"Quit", optionStyle)){
            if(toogle_sound)
audio.PlayClipAtPoint(clickAudio,transform.position, 1);
            Application.Quit();
            }
        DrawOutline(Rect(button_pos_x, button_pos_y + 130, 150, 60),
"Quit", labelStyle, Color.black, Color.white);
    }
...
```

Per ogni elemento utilizzato si specifica lo stile relativo, definito dalla variabile GUIStyle; questi ne identifica caratteristiche grafiche come

grandezza del font, immagini da utilizzare, i suoi parametri sono facilmente modificabili attraverso l'inspector, essendo queste public.
La funzione *GUI.Button* non si preoccupa solo di disegnare a schermo il bottone definito tramite la Rect, ma anche di catturare l'evento di pressione su quel dato elemento. Per questo motivo Rect è utilizzato all'interno delle if, se il bottone è premuto allora l'engine richiamerà OnGUI e quella funzione restituirà true.
DrawOutline è una funzione creata per permettere al codice di creare del testo con i bordi, che risulta essere più leggibile a schermo ed è definita semplicemente disegnando 5 diverse label:

```
public static function DrawOutline(position : Rect, text : String, style
: GUIStyle, outColor : Color, inColor : Color)
{
    var backupStyle = style;
    style.normal.textColor = outColor;
    position.x--;
    GUI.Label(position, text, style);
    position.x +=2;
    GUI.Label(position, text, style);
    position.x--;
    position.y--;
    GUI.Label(position, text, style);
    position.y +=2;
    GUI.Label(position, text, style);
    position.y--;
    style.normal.textColor = inColor;
    GUI.Label(position, text, style);
    style = backupStyle;
}
```

L'effetto finale è il seguente, l'interfaccia si adatta in base alla risoluzione:

(480x642)	Android Tall Standard 10:16
	

Gli input in Unity

Unity mette a disposizione diversi oggetti per la gestione degli Input, il principale chiamato Input è accessibile da qualsiasi game object ed è un interfaccia dell'Input System.

Questa classe è utilizzata per leggere gli "assi" impostati all'interno dell'Input Manager (*Edit>Project Settings>Input*) e per accedere ai dati del multi-touch/accelerometro per dispositivi mobile. È comune utilizzare la funzione Input.GetAxis per ogni tipo di movimento poiché questa restituisce valori "puliti" e configurabili che possono essere mappati su tastiera, joystick o mouse. Si utilizzano invece funzioni del tipo Input.GetButton per azioni relative ad eventi. Da notare bene è che ogni input non viene resettato fino alla fine del rendering del frame. È dunque buona norma occuparsi degli input solo all'interno della funzione Update.

Gestione degli input del mouse

Nel progetto ci siamo preoccupati di gestire l'input del mouse per l'inserimento dei vertici di controllo nella scena, questo sistema realizzato per la versione PC differisce da quello utilizzato per la versione mobile che vedremo di seguito.

Vogliamo inserire un punto nella scena alla pressione del tasto sinistro del mouse, in base alla posizione del puntatore, vediamo di seguito lo script *SetupController* associato all'oggetto *ControlCenter* :

```
function Update () {
    if(_active)
    {
        #if UNITY_STANDALONE_WIN
        //Se il tasto sinistro del mouse è premuto
        if(Input.GetMouseButtonDown(0))
        {

            var mousepoint : Vector3 = Input.mousePosition;
            //Escludo la parte dello schermo occupata dalla GUI
            if(mousepoint.x > 215 && mousepoint.y < 97)
                return;
            if(mousepoint.x < 35 && mousepoint.y > 470)
                return;

            checkAndMovePoint();
            insertPointInSpace(mousepoint);
        }
        #endif
}
```

Input.mousePosition ritorna esattamente il Vector3 relativo alla posizione del mouse sullo schermo, è possibile passare dalle coordinate dello schermo a quelle della scena attraverso la view della main camera, di seguito vediamo la funzione insertPointInSpace che applica questo tipo di trasformazione usando la funzione *ScreenToWorldPoint*.

```
//Prende in input le coordinate dello schermo
```

```
function insertPointInSpace(sCoord : Vector3)
{
    if(movingPointInd != -1)
        return; //sto spostando un punto
    if(ins_point_num >= point_num)
        return; //massimo raggiunto

    var pos : Vector3 = Camera.main.ScreenToWorldPoint(sCoord);

    pos.z = pointVec[0].z; //stessa profondit□el razzo

    //Controllo sulla distanza, non inserisco se troppo vicino ad un
altro punto
    for(var i = 0; i <= ins_point_num; i++){
        var dist : float = Vector3.Distance(pos, pointVec[i]);
        if(dist < min_dist)
            return;
    }

    pointVec[ins_point_num + 1] = pos;

    pointObjVec[ins_point_num] = Instantiate (pointObj,
pointVec[ins_point_num + 1], Quaternion.identity);
    ins_point_num ++;
    Camera.main.SendMessage("notifyNumPoints", point_num -
ins_point_num);
    //effettuo il report dei points
}
```

L'array pointVec è utilizzato per mantenere le posizioni relative ai vertici di controllo, sarà successivamente passato allo script che si occupa di formare la spline.

L'input del mouse è stato utilizzato anche nella gestione della telecamera, di preciso la rotella del mouse regola lo zoom, trascinare con il tasto destro del mouse invece sposta la telecamera.
Commentiamo il codice relativo allo script *CameraController* :

```
private var dragOrigin : Vector3;
private var pointer_mode : int = 2;
//0 - > null
//1 - > drag
//2 - > set
```

```
function dragCamera()
{
    var pos : Vector3;
    #if UNITY_STANDALONE_WIN
    if(Input.GetMouseButtonDown(1))
    {
        dragOrigin = Input.mousePosition;
        pointer_mode = 1;
        return;
    }

    if(!Input.GetMouseButton(1)){
        pointer_mode = 2;
        return;
    }
    pos = Camera.main.ScreenToViewportPoint(Input.mousePosition -
dragOrigin);
    #endif

    var move : Vector3 = new Vector3(pos.x * drag_speed, pos.y *
drag_speed, 0);
    move = move * -1; //inverto il movimento
```

```
    // Gestione dei limiti

    if(move.x > 0 && !isWithinBorderRight())
        move.x = 0;
    if(move.x < 0 && !isWithinBorderLeft())
        move.x = 0;
    if(move.y > 0 && !isWithinBorderTop())
        move.y = 0;
    if(move.y < 0 && !isWithinBorderBottom())
        move.y = 0;

    transform.Translate(move, Space.World);
}
```

La funzione appena vista è richiamata da Update ad ogni frame ed è utilizzata per poter spostare la telecamera sul piano xy nella scena rispettando alcuni limiti di altezza e larghezza. Quando l'utente preme il tasto destro del mouse il codice salva in una variabile temporanea la posizione a schermo del puntatore, successivamente finche il tasto resta premuto ci calcoliamo un vettore direzione tramite la differenza tra il precedente punto nello spazio e la posizione corrente del puntatore nello spazio. Infine si utilizza la direzione ottenuta per muovere la telecamera, facendo attenzione ad annullare le opportune componenti se superati i limiti del livello.

```
function zoomCamera()
{
    var fov : float = Camera.main.orthographicSize;
    var inp : float;

    #if UNITY_STANDALONE_WIN

    inp = Input.GetAxis("Mouse ScrollWheel");
```

```
    #endif
    fov += inp * zoom_speed;
    fov = Mathf.Clamp(fov, min_zoom, max_zoom);
    Camera.main.orthographicSize = fov;

    //Se sono uscito fuori dai bordi, velocemente ci rientro
    #if UNITY_STANDALONE_WIN
    if(fov != max_zoom)
        if(inp != 0 && !isWithinBorder()){
            var vect : Vector3 = transform.position -
Vector3(0,0,transform.position.z);
            vect = vect.normalized;
            transform.Translate(vect * 1.2, Space.World);
            }
    #endif
}
```

La funzione relativa allo zoom risulta essere abbastanza intuitiva per l'implementazione PC. È necessario soltanto ottenere l'input relativo alla rotella del mouse ed utilizzarlo per incrementare/decrementare la dimensione della vista ortografica della nostra telecamera, si utilizza Clamp della classe Mathf per restringere i valori della variabile in un intervallo predefinito. L'ultima parte di codice evita di fuoriuscire dai bordi del livello quando si effettua uno zoom out in prossimità dei limiti del livello.

Gestione degli input da touchscreen

Vediamo di seguito come è possibile implementare le precedenti funzioni sostituendo la parte di codice relativa a *UNITY_STANDALONE_WIN* con *UNITY_ANDROID*.

Gestire i tocchi o le gestures tramite Unity risulta essere più complicato rispetto all'uso degli input da tastiera o mouse, tuttavia l'utilizzo degli oggetti di tipo *Touch* facilita di molto l'implementazione.
Nello script *SetupController* si è implementata la parte di codice che permette di inserire un punto nello spazio tramite una pressione prolungata del tocco sullo schermo:

```
function Update () {
    if(_active)
    {
        #if UNITY_ANDROID
        if (Input.touchCount == 1){

            if(Input.touches[0].position.x > 215 &&
Input.touches[0].position.y < 97)
                return;
            if(Input.touches[0].position.x < 35 &&
Input.touches[0].position.y > 470)
                return;

            lastTouchedPoint = Input.touches[0].position;
            if(Input.anyKeyDown)
                checkAndMovePoint();

            //Inserimento con longpress
            if(Input.touches[0].phase == TouchPhase.Stationary)
            {
                longPressTimer += Time.deltaTime;
                if(longPressTimer > 1)
                {
                    longPressTimer = 0;
                    insertPointTouch();
                }
            }
```

```
            else
                longPressTimer = 0;
            }
        #endif
    }
```

```
function insertPointTouch()
{
    insertPointInSpace(lastTouchedPoint);
}
```

Da Input è possibile richiamare un array *touches* formato da elementi di tipo Touch. Questi viene aggiornato automaticamente dall'engine ad ogni frame e contiene informazioni sui tocchi attualmente attivi sul dispositivo, una volta ottenuto il tocco primario è possibile accedere alla sua "fase", ovvero in quale stato si trova il nostro gesto (Began, Ended, Stationary, Moved ecc) e di conseguenza capire se è fermo in un punto. Per fare il detect del longpress si utilizzano quindi TouchPhase.Stationary e un Timer.
Per quanto riguarda la gestione della telecamera tramite i gesti, vediamo come è possibile spostare la telecamera semplicemente trascinando il dito sullo schermo.
Nello script *CameraController* abbiamo:

```
function dragCamera()
{
    var pos : Vector3;
    #if UNITY_ANDROID
    if(Input.touchCount == 1)
    {
        pos = Input.touches[0].deltaPosition;
        if(pos.magnitude < 1.5) //lo spostamento deve essere più intenso
di un valore prestabilito
            pos = Vector3(0,0,0);
        else
```

```
            pointer_mode = 1;
        pos *= 0.085; //diminuisco la sensibilità
    }
    else
        pos = Vector3(0,0,0);
    if(Input.touchCount == 0)
        pointer_mode = 2;
    #endif

    var move : Vector3 = new Vector3(pos.x * drag_speed, pos.y *
drag_speed, 0);
    move = move * -1; //inverto il movimento

    // Gestione dei limiti

    if(move.x > 0 && !isWithinBorderRight())
        move.x = 0;
    if(move.x < 0 && !isWithinBorderLeft())
        move.x = 0;
    if(move.y > 0 && !isWithinBorderTop())
        move.y = 0;
    if(move.y < 0 && !isWithinBorderBottom())
        move.y = 0;

    //
    transform.Translate(move, Space.World);
}
```

Una volta verificato che sullo schermo è presente un solo tocco, possiamo accedere alla variabile di tipo Touch ad esso relativo, dalla quale possiamo ricavare la proprietà deltaPosition che restituisce la variazione di posizione rispetto al precedente frame. L'implementazione è quindi semplificata da deltaPosition che rappresenta esattamente la direzione che vogliamo dare

allo spostamento della Main Camera, utilizziamo le solite funzioni per impedire che questa fuoriesca dai limiti imposti per il livello.
Più complicato è il discorso riguardo allo zoom, poiché la tecnica pinch to zoom impone l'utilizzo di due tocchi, l'idea è quella di utilizzare la variazione della distanza tra i due tocchi per aumentare/diminuire lo zoom della telecamera:

```
#if UNITY_ANDROID
var lastVal : float;
#endif

function zoomCamera()
{
    var fov : float = Camera.main.orthographicSize;
    var inp : float;

    #if UNITY_ANDROID
    if (Input.touchCount == 2) {
        //calcolo la distanza dei due tocchi

        var dist : float = (Input.touches[0].position -
Input.touches[1].position).magnitude;

        if(lastVal != 0){
            var diff : float = lastVal - dist;
            if(diff > 0.5)
                inp = 0.08;
            if(diff < -0.5)
                inp = -0.08;
        }

        lastVal = dist;
        pointer_mode = 1;
    }
```

```
    else
    {
        inp = 0;
        lastVal = 0;
    }
    #endif

    fov += inp * zoom_speed;
    fov = Mathf.Clamp(fov, min_zoom, max_zoom);
    Camera.main.orthographicSize = fov;
}
```

Il codice soprastante fa uso di una variabile d'appoggio lastVal per mantenere il valore della distanza tra i due tocchi nel frame precedente: facendo un confronto tra questa distanza e quella del frame attuale possiamo capire se aumentare o diminuire lo zoom della telecamera di un fissato valore.

Eventi

Per approfondire lo scripting in Unity si è pensato di spiegare come sono state implementate determinate azioni all'interno del progetto. Un evento consiste in azioni che si susseguono nell'engine al verificarsi di una determinata condizione, quale lo scontro tra due oggetti, la pressione di un tasto, l'esplosione del razzo.

Lancio del razzo

Ad esempio vediamo cosa avviene quando viene premuto il pulsante LAUNCH durante la fase di setup, diversi script sono coinvolti nell'evento:

```
//In OnGUI()
if(num_points == 0){
                if(GUI.Button(Rect(originalWidth - _width, _height,
_width, 75), "LAUNCH", optionStyle)){
                    if(game_type == 1)
                        game_mode = 2; //space
                    if(game_type == 0)
                        game_mode = 1; //land
                    launchRocket();
                        }
            }
//////
```

All'interno di OnGUI è gestita la chiamata alla pressione del bottone LAUNCH, quando questa avviene la modalità di gioco è cambiata impostando una costante alla variabile *game_mode*, successivamente si chiama la funzione *launchRocket*.

```
public function launchRocket()
{
//mando in play un AudioClip se dal MusicCenter l'audio è attivo
    if(GetComponent(MusicCenter).toogle_sound)
      audio.PlayClipAtPoint(launchAudio,transform.position, 1);

    controlCenter.SendMessage("launchRocket");
    Camera.main.GetComponent(CameraController).followRocket = true;
    controlCenter.SendMessage("setActive",false); //disattivo inserimento
}
```

Questa breve funzione si occupa di portare l'applicazione nella fase di lancio richiamando diverse funzioni e settando qualche parametro, si dice alla telecamera di iniziare a seguire il razzo impostando a true una variabile dello script *CameraController*, si disattiva l'inserimento dei punti passando

false alla funzione setActive di SetupController e si chiama la funzione launchRocket sempre presente in quest'ultimo script. Quello che abbiamo visto fino ad ora è presente nello script Interface che si occupa della GUI, assegnabile a qualsiasi oggetto che non venga distrutto nella scena.

```
public function launchRocket()
{
    SendMessage("instantiateSpline",pointVec);
    rocketObj.GetComponent(FollowSpline)._active = true;
}
```

Dal controlCenter richiamo la funzione *instantiateSpline*, che crea il percorso del razzo attraverso i punti definiti nel vettore pointVec. Dal riferimento all'oggetto che rappresenta il razzo invece accedo allo script FollowSpline e setto a true la variabile _active, che lo fa muovere lungo la curva. Come si è visto si tratta tutto nel sapere come interagire tra script e oggetti diversi.

Esplosione del razzo

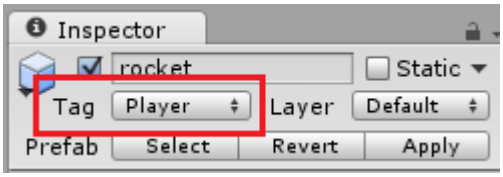
Lo script RocketExplosion è quello che si occupa dell'evento di esplosione del razzo, il quale avviene ogni volta che questi entra in contatto con un oggetto "solido", come il terreno o un ostacolo. Per catturare l'evento di collisione di due corpi si utilizza la funzione OnTriggerEnter vista in precedenza, questo script dunque deve essere assegnato ad ogni oggetto che vogliamo sia il Trigger del nostro evento, ricordandoci che necessita di un Collider con la checkbox isTrigger settata a true.

```
public var isSafeSpot : boolean = false; //se impostata a true allora è una zona sicura in fase di landing

function OnTriggerEnter(other : Collider)
```

```
{
    var rocketLanding : GameObject;
    player = other.gameObject;
    if(other.tag == "Player" || other.tag == "player" || other.tag ==
"rfall")
    {
        if(!isSafeSpot){
            destroyRocket();
            rocketLanding = GameObject.FindGameObjectWithTag("rfall");
            if(rocketLanding != null)
                Destroy(rocketLanding); //distruggo il razzo con il
paracadute
        }
        else
        {
            //Controllo di essere in modalità landing
            rocketLanding = GameObject.FindGameObjectWithTag("rfall");
            if(rocketLanding == null)
                destroyRocket();
            else
            {

rocketLanding.Find("paracadute").animation.Play("para_close",
PlayMode.StopAll);
                Destroy(rocketLanding.GetComponent(Rigidbody)); //no
collision
                //landing riuscito, lo comunico
                Camera.main.SendMessage("levelComplete");
            }
        }
    }
}
```

Questa parte di codice si occupa anche di definire quali sono le zone sicure (come il terreno) utilizzate per l'atterraggio del razzo nella land mode.

Come vediamo, l'evento è catturato tramite l'utilizzo di OnTriggerEnter (meno dispendioso rispetto a OnCollisionEnter), la variabile in ingresso other rappresenta il collider che ha effettuato la collisione, da questi è possibile risalire al gameobject e capire attraverso un tag se si tratta del giocatore (ovvero il razzo). I tag sono uno strumento molto importante all'interno di Unity, si definiscono aprendo la finestra Tag dall'inspector e sono utilizzati per assegnare una stringa ad un oggetto, utile nei casi come il nostro, in cui abbiamo bisogno di capire con quale oggetto abbiamo a che fare.



La funzione *destroyRocket* non elimina effettivamente il razzo dalla scena, si occupa invece di renderlo impercettibile ad essa richiamando *hideRocket* che si occupa di disabilitare il renderer e il collider dell'oggetto, in questo modo potremo riutilizzarlo dopo il reset della scena.

```
private function destroyRocket()
{
            //istanzio l'animazione dell'esplosione
            Instantiate(animObj, player.transform.position,
Quaternion.identity);
            //rendo invisibile il razzo
            GameObject.Find("rocket").SendMessage("hideRocket");
            //riferisco all'Interface che ho fallito il livello
            Camera.main.SendMessage("levelFailed");
            Camera.main.GetComponent(CameraController).followRocket =
false;
}
```

Dati persistenti

Può capitare che al verificarsi di un evento sia necessario dover salvare dei dati dell'applicazione in memoria secondaria, in modo che questi siano

accessibili anche dopo che questa venga chiusa. Si vuole quindi creare un file di salvataggio, una tecnica comunemente utilizzata anche per comunicare tra le diverse scene del gioco, essendo queste isolate tra loro.
Nel progetto lo script *MusicCenter* si occupa di salvare in questo modo i dati relativi al sonoro, così facendo il giocatore non dovrà ogni volta accedere alle impostazioni per disattivare/attivare l'audio all'avvio di ogni scena o nuova sessione di gioco.

```
import System.IO;

public var toogle_sound = true;
public var toogle_music = true;
private var path_ : String;

function Start () {
    path_ = Application.persistentDataPath + "/" + "tmp/";
    if(!System.IO.Directory.Exists(path_))
        System.IO.Directory.CreateDirectory(path_);
    if(!System.IO.File.Exists(path_ + "options.gme")){
        saveSoundState();
    }
    loadSoundState();
}

public function set_music( val : boolean )
{
    //Attivo o disattivo l'audiosource
    toogle_music = val;
    if(!toogle_music)
        GetComponent(AudioSource).mute = true;
    else
        GetComponent(AudioSource).mute = false;

    saveSoundState();
}
```

```
public function set_sound( val : boolean )
{
    //semplicemente controllo questo valore
    //quando devo mandare in play un AudioClip
    toogle_sound = val;
    saveSoundState();
}

public function loadSoundState()
{
    if (System.IO.File.Exists(path_ + "options.gme"))
    {
        var sRead = new File.OpenText(path_ + "options.gme");
        toogle_sound = parseBool(sRead.ReadLine());
        toogle_music = parseBool(sRead.ReadLine());
        sRead.Close();

        set_music(toogle_music);
        set_sound(toogle_sound);
    }
}

public function saveSoundState()
{
    var sWrite : StreamWriter = new StreamWriter(path_ + "options.gme");

    sWrite.WriteLine(toogle_sound);
    sWrite.WriteLine(toogle_music);
    sWrite.Flush();
    sWrite.Close();
}
```

Le funzioni set_music e set_sound sono richiamate dagli script esterni ogni volta che il giocatore attiva o disattiva l'audio, queste richiamano *saveSoundState* che si preoccupa di salvare su un file attraverso l'oggetto

StreamWriter il valore delle variabili toogle_sound e toogle_music. Ogni volta che l'oggetto a cui è associato MusicCenter viene caricato in memoria, ovvero all'avvio di ogni scena, dalla funzione Start() richiamiamo loadSoundState che va a leggere i valori salvati nel file options.gme.

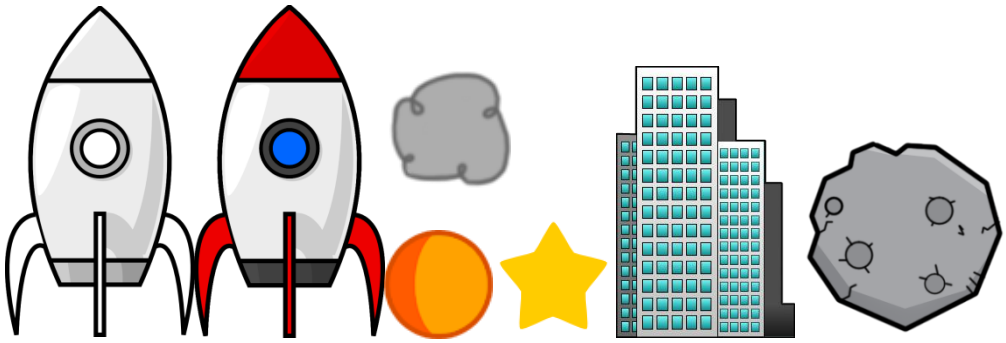
Capitolo 4

Aspetto grafico

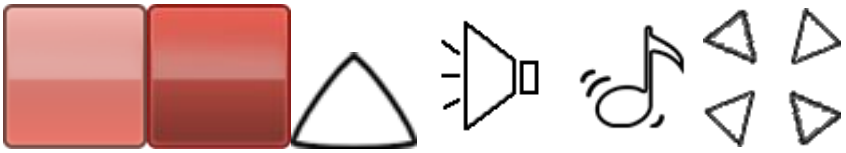
Nei videogiochi moderni si fa sempre più attenzione alla componente grafica, la quale diventa un importante parte dello sviluppo e non deve di conseguenza essere trascurata.

Prima di tutto dopo aver definito l'idea si deve iniziare a pensare a quale si vuole che sia lo stile grafico del progetto. Per stile grafico si intende non solo la qualità delle immagini da utilizzare, ma anche quanto si vuole essere precisi nei dettagli, scegliendo tra una grafica più realistica invece che una più da cartone animato, importante risulta essere anche la scelta dei colori, cercando sempre di rimanere in uno schema ben preciso, evitando forti contrasti. Per Rocket Spline si è scelta una grafica a due dimensioni, più adatta ad un gioco mobile perché più leggera, lo stile adottato è quello cartoon con colori accesi e tratti ben definiti, di rado sono applicate luci e riflessi sulle immagini per dare un effetto lucentezza, sono state invece poco usate le ombre, non adatte allo stile scelto. Prima di iniziare a lavorare sulla grafica deve essere già ben chiaro qual è l'aspetto che si vuole dare al gioco, cercando quanto è più possibile di mantenere un certo stile, creare oggetti e forme troppo differenti da esso porta all'aggiunta di elementi di disturbo, ne può bastare uno per compromettere l'intero lavoro sulla grafica.

A seguito di una fase di ricerca si è preso spunto da un immagine di un razzo proposta dal sito clipartist.net[7], successivamente si è utilizzato un programma di grafica (*adobe photoshop*) per definirne diversi aspetti e colori, una volta ottenuto un riferimento si è cercato di definire tutte le altre componenti quali il terreno, il cielo o le asteroidi rifacendoci sempre allo stile grafico di quest'immagine.

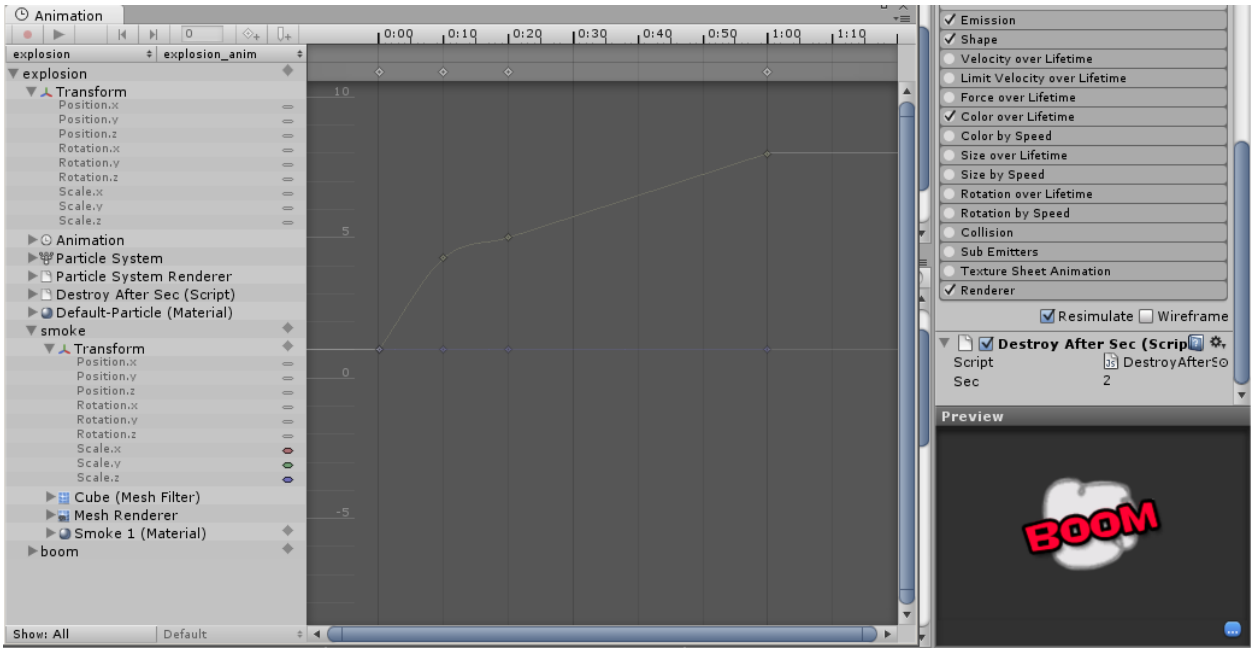


In questo modo si è creato l'aspetto del gioco, passando da programmi per la gestione delle immagini come adobe photoshop, si sono create diverse componenti grafiche stilizzate, le quali conferiscono al progetto uno stile cartoon. Per quanto riguarda l'interfaccia grafica si è anche per questa scelto un aspetto semplice e non distaccato dal resto del gioco, il font e le icone sono di colore bianco con contorno nero, in modo da essere in risalto al giocatore.



Animazioni

Unity permette di definire delle animazioni che controllano le proprietà dei componenti nel tempo, di conseguenza modificando a piacere la componente Transform di un gameobject si possono ottenere delle semplici trasformazioni. La gestione delle animazioni avviene tramite una finestra dedicata (*Window>Animation o Ctrl+6*) ed è abbastanza intuitiva, un animazione può coinvolgere diversi oggetti figli del gameobject a cui è assegnata.

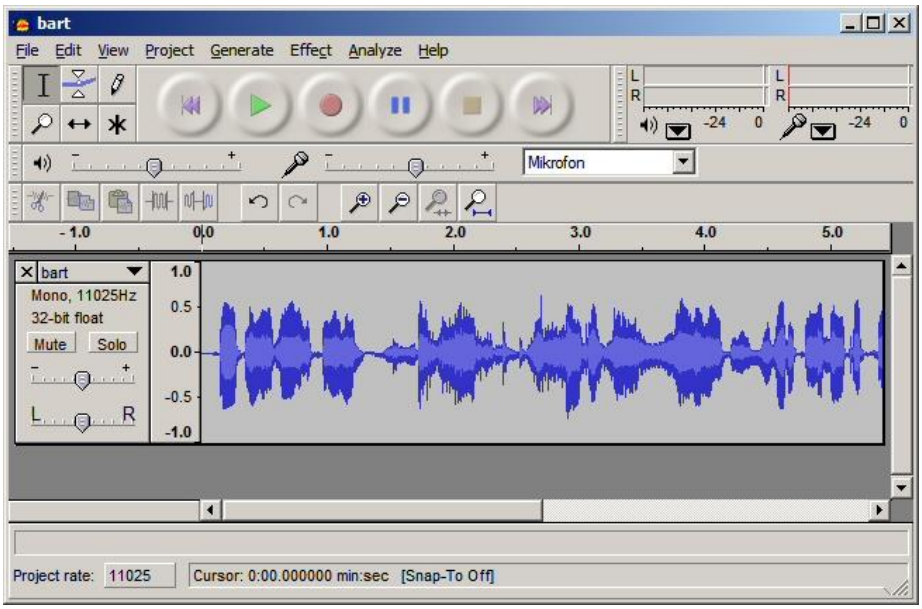


Un idea per gestire animazioni quali esplosioni o “effetti istantanei” è quella di istanziare un oggetto che si occupa solo dell’animazione e che si rimuove dalla scena al suo termine. Naturalmente cercando di mantenere un certo stile grafico anche le animazioni non posso distaccarsi da quello che dovrà essere il risultato finale, poiché ci rifacciamo ad un ambiente cartoon le animazioni sono formate da immagini stilizzate rappresentati effetti di fumo e luce, non sono utilizzate particelle che potrebbero dare un effetto troppo realistico.

Sonoro

Il sonoro comprende sia gli effetti audio che sono ascoltabili al verificarsi di un evento e sia la musica di fondo che serve a dare un determinato tono all’ambiente di gioco. Anche questa componente è importante nello sviluppo di un videogame, non è facile scegliere quali sono gli effetti sonori più adatti ad un ambiente e ricrearli autonomamente richiede

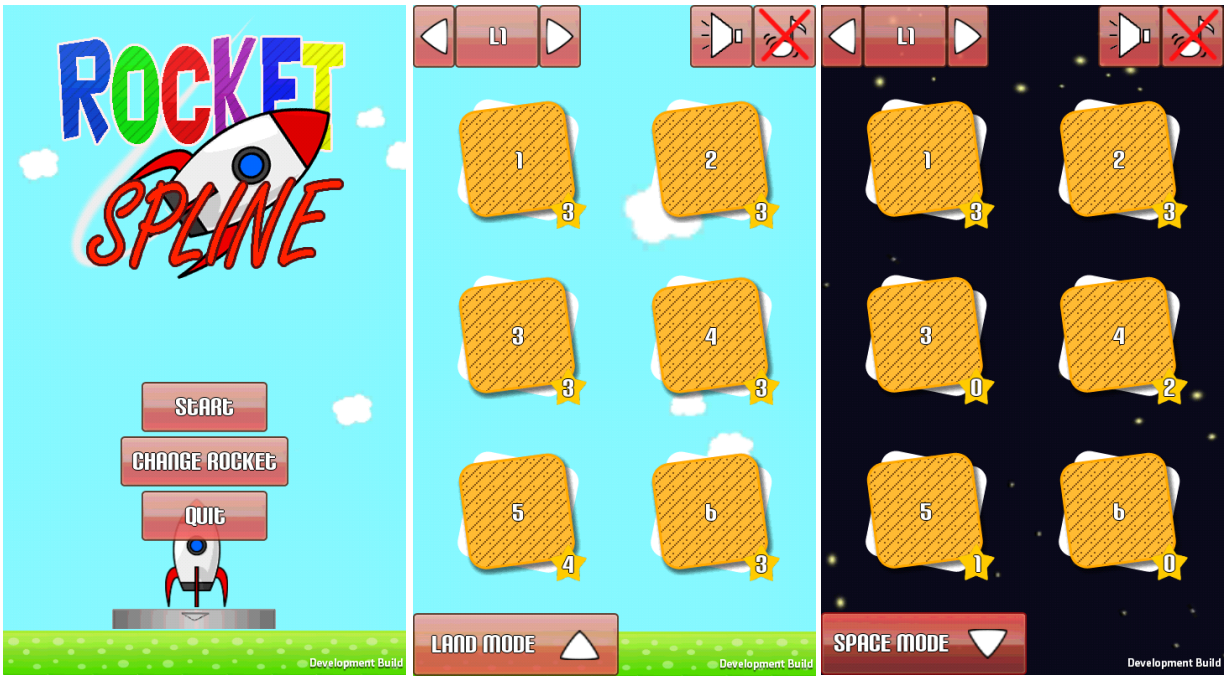
attrezzature che vanno oltre ad un semplice microfono. Sul web diverse community mettono a disposizione file audio a supporto di progetti amatoriali, per Rocket Spline si vuole dare merito a *opengameart.org*[8]. Diversi file audio necessitano di tagli o modifiche in generale, un programma gratuito per effettuare tali operazioni è *Audacity*[9], grazie al quale si possono anche salvare i propri file in formato .ogg.



Unity mette a disposizione un potente componente chiamato *AudioSource* per la gestione degli AudioClip nella scena, come detto in precedenza questo si occupa di simulare l'ascolto come sarebbe quello di un utente nella scena, tenendo conto di distanze e dell'effetto doppler. Poiché Rocket Spline si rifà ad un ambiente in 2D delle modifiche sono state adottate per rendere l'AudioSource della scena il più semplice possibile, in modo da non tener conto delle distanze tra ascoltatore e origine del suono.

Risultato Finale

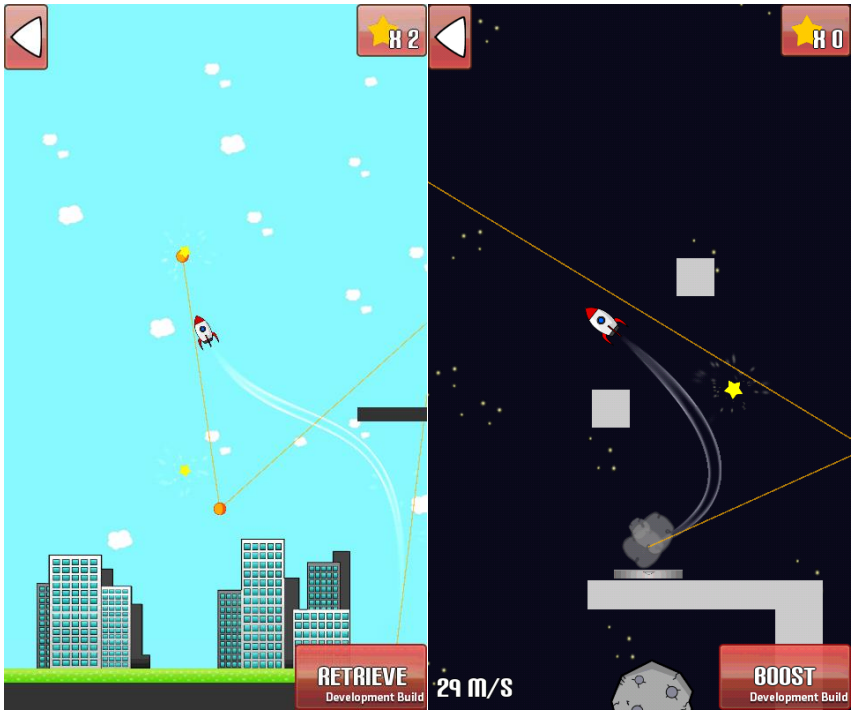
Di seguito una serie di screenshot che illustrano le schermate principali di Rocket Spline.



Lanciando l'applicazione, a seguito del logo di Unity, il gioco presenterà la schermata principale. Inizialmente saremo di fronte al logo del gioco e a tre pulsanti, il primo permetterà di passare alla schermata successiva, il secondo di cambiare l'aspetto del razzo e il terzo di uscire dall'applicazione. La successiva schermata è quella della scelta del livello da giocare, tramite un pulsante in basso è possibile passare da una modalità di gioco all'altra. In questa schermata si possono usare i tasti in alto a destra per gestire l'audio, mentre i tasti in alto a sinistra sono utilizzati per scorrere la lista dei livelli. Ogni livello deve essere sbloccato e non è giocabile prima di aver terminato il precedente livello. Se un livello è bloccato, un'icona a forma di lucchetto apparirà al posto del numero del

livello. In basso a destra ad ogni pulsante dei livelli è presente un numero che indica il numero di stelle ottenute in quel livello. Premendo su uno dei sei quadrati centrali nella schermata si passa alla scena che contiene l'effettivo livello scelto.

Delle animazioni sono richiamate tra una schermata e l'altra in modo da dare un effetto di continuità tra i menù.



Una volta scelto il livello ci ritroveremo di fronte a due diversi ambienti a seconda della modalità di gioco scelta.

Nella LAND MODE il giocatore cercherà di raccogliere tutte le stelle in cielo mentre evita gli ostacoli presenti sul campo. Un pulsante in basso a destra apparirà per permettere al giocatore di far atterrare il razzo in una zona sicura a seguito del lancio.

Nella SPACE MODE il giocatore userà la velocità del razzo per distruggere le asteroidi presenti nello spazio. In basso a destra un pulsante permetterà di raddoppiare per un breve istante la velocità del razzo, la quale è indicata

in basso a sinistra.
Comune ad entrambe le modalità è un tasto presente in alto a sinistra che permette di uscire dal livello e modificare le impostazioni audio.
Il livello termina con una schermata che indica il numero di stelle ottenuto, se sono soddisfatti gli obiettivi della modalità di gioco scelta, un pulsante apparirà per proseguire con il livello successivo.

Conclusione

Il lavoro dietro un videogioco risulta essere lungo e a volte difficile. Infatti per produrre un prodotto di qualità sono necessarie doti che vanno ben oltre la logica della semplice programmazione. Si tratta principalmente di inventiva, capacità artistiche e sonore, conoscenza del campo e tanto tempo libero. Quanto abbiamo appena visto nella tesi è solo parte del lavoro dietro a Rocket Spline, questa parte risulta tuttavia essere quella più interessante a chi si affaccia per la prima volta a questo ambiente, il lavoro sulla grafica, così come quello sul sonoro sono importanti, ma sono molto più intuitivi e possono variare in base allo stile scelto per il gioco dal suo sviluppatore.

Per capire come funziona il gioco inizialmente abbiamo parlato di curve spline, la tecnica matematica sulla quale si basa il gameplay dell'intero progetto, ci siamo soffermati su come sia possibile definire una curva in un piano, prima discutendo della tecnica di interpolazione e successivamente quella di approssimazione con nodi multipli, che risulta essere poi quella implementata.

Abbiamo spiegato come è possibile definire una funzione polinomiale attraverso le basi b-spline, senza trascurare le importanti proprietà che ne derivano e che sono poi sfruttabili nel gioco se conosciute. Per approfondire l'argomento si è parlato di curve razionali e spline nello spazio 3D, argomenti attualmente molto comuni nel campo della computer grafica.

Per comprendere pienamente il lavoro dietro Rocket Spline è stato necessario introdurre Unity3D, l'ambiente di sviluppo scelto per l'applicazione, spiegando prima brevemente le sue caratteristiche e l'interfaccia e successivamente lo scripting, discutendo su quali sono le classi e i concetti comunemente usati per lo sviluppo di un videogame.

Infine ci siamo soffermati sul progetto, cos'è Rocket Spline, come funziona, quali scelte implementative sono state effettuate durante la sua produzione. In quest'ultima parte abbiamo parlato di come sono state implementate le curve spline attraverso l'uso dell'algoritmo di De-Boor e di come sia possibile muovere un oggetto su di esse. Si è inoltre discusso di come è stata gestita la Graphical User Interface (GUI) e di come si utilizza la classe Input per rendere la nostra applicazione interattiva. In generale non si è voluto semplicemente mostrare come sono state implementate alcune particolarità del progetto, ma anche fornire informazioni sul loro funzionamento, in modo da poter dare più informazioni possibili sullo scripting in Unity.

Importante ai fini del progetto è saper pianificare il lavoro, se il team di sviluppo è composto da poche persone, o in questo caso da un singolo, si deve da subito pensare quali feature della nostra idea conviene implementare, tempi di lavoro troppo lunghi possono compromettere la riuscita del lavoro. La grafica del progetto risulta essere molto semplice, si rifà ad uno stile cartoon, una volta scelto lo stile grafico è importante mantenerlo, un qualsiasi elemento di disturbo potrebbe compromettere la qualità dell'intero lavoro, è importante mantenere un certo equilibrio tenendo sempre a mente quale vogliamo sia il risultato finale in base alle nostre doti artistiche.

In conclusione si è voluto presentare il lavoro dietro allo sviluppo di Rocket Spline, un puzzle game basato sulle curve spline. Si spera che le informazioni all'interno della tesi siano sufficienti a chi si approccia per la prima volta allo sviluppo di un applicazione con Unity o di chi semplicemente vuole approfondire le sue conoscenze sulle curve spline, ma soprattutto si vuole dare l'idea della quantità di lavoro dietro un videogioco, sia questo semplice o complesso, sperando di suscitare curiosità per questo stimolante ambiente di lavoro.

Bibliografia

[1] J. Hoschek – D. Lasser,Computer Aided Geometric Design, A.K. Peters Wellesley Massachussets 1993.

[2][3] Piegel W.Tiller, The NURBS Book – II Edition, Springer Verlag 1997

[4] Farin, Curves and Surfaces for CADG V Edition, Morgan Kaufmann Publishers

Webgrafia

[5] Unity Game Engine
http://unity3d.com/

[6] Unity Scripting Reference
http://docs.unity3d.com/Documentation/ScriptReference/

[7] Clip Artist .net
http://clipartist.net/

[8] Open Game Art
http://opengameart.org/

[9] Audacity
http://audacity.sourceforge.net/