UNIVERSITÀ DEGLI STUDI DI MILANO

Facoltà di Scienze Matematiche, Fisiche e Naturali

Corso di Laurea in Informatica (Crema)

Studio comparativo di software per la simulazione dinamica di strutture

RELATORE
Dott. Gabriele Gianini

TESI DI LAUREA DI
Fabio Cometti
Matr. 671979

Anno Accademico 2006/2007

Ringraziamenti

La stesura della tesi di laurea rappresenta l'ultimo passo di un lungo cammino. Quando si arriva in fondo è inevitabile guardarsi indietro ed osservare la strada percorsa; altrettanto inevitabile è l'accorgersi di quante persone ci sono state vicine durante il nostro viaggio, proprio per questo mi pare doveroso ringraziare tutti coloro che in un modo o nell'altro mi hanno aiutato a raggiungere questo traguardo; in particolare desidero ringraziare:

- *Laura, per avermi spinto e incoraggiato ad affrontare quella meravigliosa sfida che è l'università.*
- *I miei genitori, per il sostegno dimostrato in ogni momento e per tutto il tempo che hanno perso a sbrigare le pratiche burocratiche che io non potevo assolvere.*
- *Francesco, per essermi stato vicino nel momento più difficile di questi tre anni e più in generale per essermi vicino da sempre.*
- *Il mio relatore, Il Dottor Gianini, per l'aiuto e la disponibilità dimostrate in ogni momento.*
- *Giovanni e Stefano, i miei colleghi di lavoro, per aver spesso testato i miei progetti sui loro computer a loro rischio e pericolo.*
- *I ragazzi addetti alla registrazione delle video lezioni: forse non vi rendete conto di quanto è prezioso il vostro contributo per chi studia e lavora.*
- *Il mio datore di lavoro, il Signor Alfredo Fusaglia, per la pazienza dimostrata e per aver chiuso un occhio sulle tante ore di permesso chieste.*
- *Luca e Chiara per la loro amicizia.*
- *I ragazzi del calcetto: Andrea, Carlo, Gabo, Moro, Cesco, Pier, Giorgio, Natale, Fulvio, Gigi, Silvano, Becca, Paolo, Mitchell e tutti gli altri, per avermi aiutato a distrarmi nei periodi di studio intenso.*
- *Tutti coloro che in qualche modo mi hanno aiutato in questa bellissima avventura.*

Fabio Cometti

Indice

Indice delle figure

Introduzione

Spesso l'adozione di un prodotto software da parte di un'azienda può rappresentare un problema: da una parte vi sono i prodotti commerciali, costosi ma ben supportati dai propri produttori; dall'altra vi sono i prodotti Open Source, spesso gratuiti ma a volte mal supportati dai programmatori. Per chi decidesse comunque di adottare una soluzione Open Source si pone un altro problema che spesso può essere addirittura di più difficile soluzione rispetto a quello del supporto: la scelta del giusto prodotto software. Spesso ci si può trovare davanti a decine di software tutti con caratteristiche simili, ma quale si adatta veramente ai nostri bisogni? Quale ci può garantire l'effettivo supporto di cui abbiamo bisogno? Dare una risposta a queste domande non è sempre facile.

Scopo di questa tesi è quello di illustrare un metodo per la valutazione di prodotti software Open Source e in particolare per la scelta e la valutazione di un Motore Fisico *real time* da integrare in progetti di tipo ludico.

Nel capitolo uno verrà spiegato che cos'è un motore fisico, a cosa serve, da quali parti è composto e perché è utile e produttivo utilizzare prodotti di terze parti anziché creare del codice personalizzato.

Nel secondo capitolo vengono presentati i due motori fisici oggetto della nostra valutazione: *Open Dynamics Engine* e *Newton Game Dynamics.* Di ciascun motore vengono illustrate le caratteristiche principali e le funzionalità offerte.

Nel capitolo tre viene presentato il framework *Business Readiness Rating model* (BRR). BRR rappresenta il tentativo di creare un modello aperto e standard per la valutazione di manufatti software (ma non solo) Open Source.

Il quarto capitolo è invece dedicato all'applicazione del modello BRR al nostro caso di studio: verrà presentato lo scenario creato per le nostre prove, i requisiti che richiediamo al nostro motore fisico, le metriche che utilizziamo per la valutazione, la collezione di tutti i dati necessari per la stima e i calcoli che ci portano ad ottenere un risultato numerico sulla stima dei due software.

1 – La scelta di un Motore Fisico: criteri e problemi

1.1 - Cos'è un motore fisico?

1.1.1 - Definizione

*"A **physics engine** is a computer program that simulates Newtonian physics models, using variables such as mass, velocity, friction and wind resistance. It can simulate and predict effects under different conditions that would approximate what happens in real life or in a fantasy world. Its main uses are in scientific simulation and in video games."*

Un **motore fisico** è un programma per computer che simula modelli fisici Newtoniani usando variabili come massa, velocità, attrito e resistenza del vento. Esso può simulare gli effetti, sotto diverse condizioni, che approssimano quello che succederebbe nella vita reale o in un mondo fantastico. Il suo impiego principale si ha nelle simulazioni scientifiche e nei videogame.

[*Wikipedia*]

1.1.2 - Descrizione

Ci sono generalmente due classi di motori fisici, *in tempo reale* e ad *alta precisione*. I motori fisici ad alta precisione (o simulazioni dinamiche) richiedono maggiore potenza per calcolare gli effetti fisici in modo molto preciso e sono di solito utilizzati dagli scienziati o nei film di animazione realizzati al computer. Nei videogame o in altre forme di computazione interattiva il motore fisico dovrà semplificare i suoi calcoli e abbassare la sua accuratezza in modo da poter fornire un adeguata velocità di risposta in modo da non compromettere l'interattività; in questo caso parliamo di motori fisici in tempo reale. I giochi per computer utilizzano la fisica per aumentare il grado di realismo.

I motori fisici sono generalmente composti da due componenti principali: un sistema di rilevamento delle collisioni (*Collision Detection System*) e il componente per la simulazione

fisica che è responsabile per la risoluzione delle forze che agiscono sugli oggetti simulati. Esistono tre principali paradigmi di sviluppo per le simulazioni fisiche:

- I metodi delle penalità, dove le interazioni tra gli oggetti sono comunemente modellate come sistemi *massa-molla*. Questo tipo di motori è utilizzato principalmente per la fisica dei corpi morbidi e deformabili.
- I metodi basati sui vincoli, dove le equazioni dei vincoli sono risolte con una stima delle leggi fisiche.
- I metodi basati sull'impulso, dove un impulso viene applicato a ogni interazione tra due oggetti diversi.

Infine esistono metodi ibridi che possono combinare diversi aspetti di tutti e tre i paradigmi principali.

1.1.3 - Motori scientifici

Uno dei primi computer *general-purpose*, ENIAC, fu utilizzato come un tipo molto semplificato di motore fisico. Esso venne utilizzato per compilare tabelle balistiche che avrebbero aiutato l'esercito degli Stati Uniti a stimare dove sarebbero atterrate le granate di massa variabile al variare dell'angolo di lancio e della carica di polvere utilizzata tenendo anche conto della deriva causata dal vento. I risultati venivano calcolati una sola volta e poi tabulati e stampati in tabelle che venivano affidate ai comandanti dell'artiglieria.

I motori fisici sono stati comunemente usati sin dal 1980 per calcolare i flussi di venti e nubi al fine di prevedere le condizioni meteo. Questo campo di applicazione prende il nome di modello *Computational Fluid Dynamics Modeling* (modello computazionale della dinamica dei fluidi) e all'interno di esso alle varie particelle d'aria viene assegnato un vettore forza, tali vettori vengono poi combinati tra loro per vaste regioni di spazio per mostrare il movimento d'insieme di tutta la massa d'aria al fine di prevedere gli spostamenti delle zone di alta e bassa pressione. Dati i requisiti di estrema velocità e precisione richiesti per queste operazioni sono stati sviluppati appositi processori conosciuti con il nome di processori vettoriali che accelerano i calcoli.

Generalmente le previsioni del tempo sono ancora una scienza non molto accurata perché la risoluzione della simulazione dell'atmosfera non è abbastanza dettagliata per venire incontro alle condizioni del mondo reale e anche perché piccole fluttuazioni del sistema che non vengono modellate nella simulazione possono cambiare drasticamente i

risultati delle previsioni dopo parecchi giorni. Modelli simili al *Computational Fluid Dynamics Modeling* sono spesso usati anche per la progettazione di nuovi tipi di aerei e navi e possono fornire agli ingegneri informazioni che solitamente si possono ottenere solo con costosi test effettuati nel tunnel del vento.

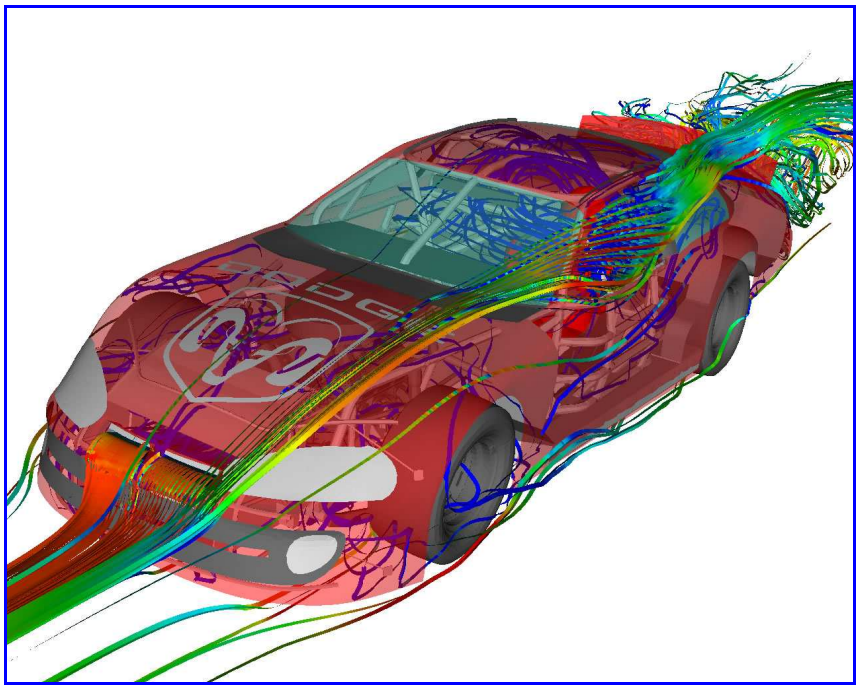

Figura 1 - Esempio di simulazione del tunnel del vento

L'industria dei pneumatici usa le simulazioni fisiche per scoprire come i nuovi tipi di gomma reagiscono sotto diverse condizioni di bagnato e asciutto introducendo variabili come la flessibilità del pneumatico oppure diversi livelli di carico.

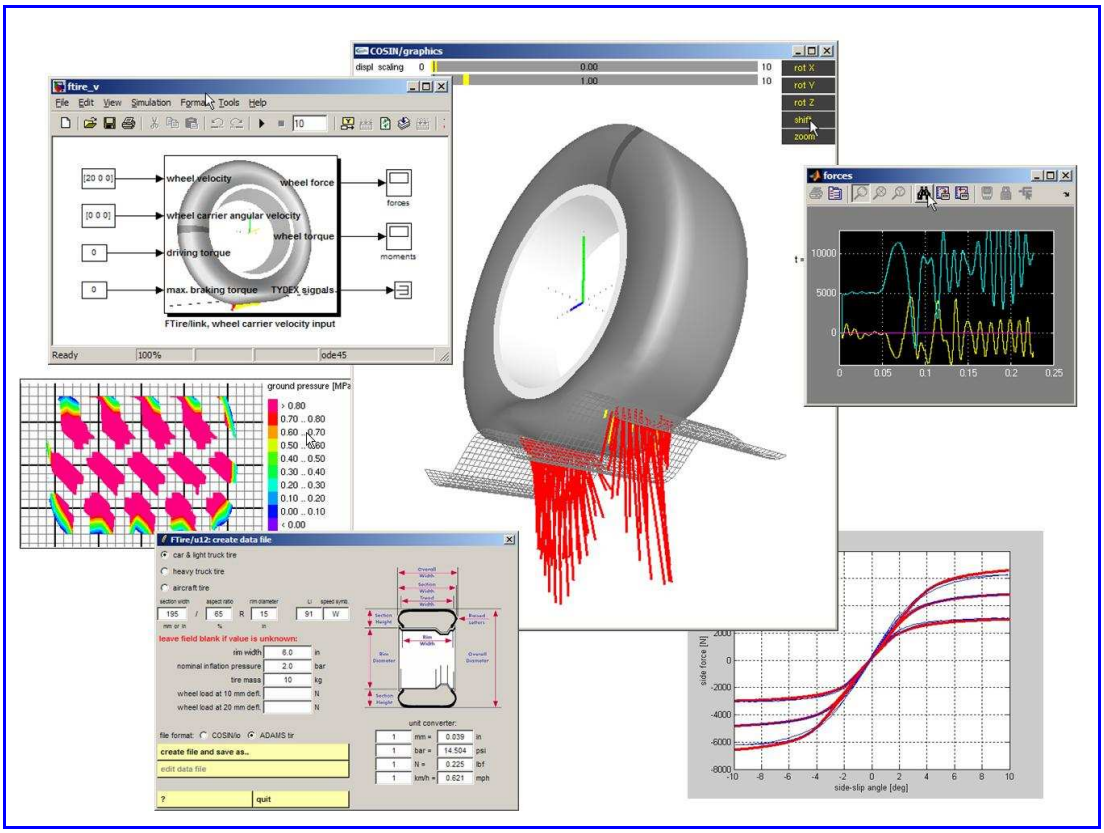

Figura 2 - Simulazione fisica utilizzata nella progettazione dei pneumatici

L'industria elettronica usa il *fluid dynamic modeling* per esaminare come si muove l'aria del raffreddamento dentro il case del computer al fine di localizzare eventuali punti che necessitano di un raffreddamento supplementare.

1.1.4 - Motori fisici per videogame

In molti videogiochi la velocità di simulazione è molto più importante dell'accuratezza della simulazione stessa. Tipicamente molti oggetti 3D in un gioco sono rappresentati da due *mesh* separate. Una di queste ha un'alta complessità e una forma molto dettagliata ed è quella che il giocatore vede durante il gioco, per esempio un vaso con curve eleganti e maniglie arrotondate. Comunque, per scopi di velocità, una seconda *mesh* invisibile e molto semplificata è utilizzata per rappresentare l'oggetto nel motore fisico. Ad esempio il vaso di cui sopra potrebbe apparire al motore fisico come nulla più che un semplice cilindro. È quindi impossibile durante il gioco sperare di poter inserire una sbarra nel vaso oppure sparare a una delle maniglie e vederla saltar via come nella realtà semplicemente perché il motore non conosce la cavità del vaso o l'esistenza delle maniglie, esso conosce solo l'esistenza di un cilindro pieno. La *mesh* semplificata usata per i calcoli fisici è spesso riferita a una geometria di collisione. Questa può essere una *bounding box*, una sfera o *convex hull*. I motori che utilizzano una *bounding box* o una sfera come forma

per il rilevamento delle collisioni sono considerati estremamente semplici. Generalmente una *bounding box* viene usata per una fase generale di rilevamento delle collisioni per ridurre il numero di possibili collisioni prima di applicare un più costoso algoritmo per il rilevamento di collisioni tra *mesh* e *mesh* in una seconda fase ristretta.

Figura 3 - Esempi di bounding box, sfera di contenimento e convex hull

Nel mondo reale le leggi della fisica sono sempre attive. C'è un costante *moto Browniano* che interessa tutte le particelle dell'universo con continue forze di attrazione e repulsione tra le particelle stesse. Per il motore fisico di un video game questa costante e sempre attiva precisione non è necessaria e sarebbe anzi una rovina per le limitate capacità della CPU. Nel mondo virtuale 3D di *Second Life* ad esempio se un oggetto è immobile o non copre una certa distanza minima in due secondi allora il calcolo della fisica per l'oggetto in questione viene disabilitato e l'oggetto stesso viene "congelato" sul posto. Esso rimane congelato fino a quando avviene una collisione con un oggetto fisicamente attivo che riattiva il processo di calcolo della fisica. Questo congelamento degli oggetti stabilmente non in movimento consente al motore fisico di risparmiare potenza di calcolo e di incrementare così il *framerate* degli altri oggetti correntemente in movimento.

Il principale limite al realismo dei motori fisici è la precisione dei numeri che rappresentano la posizione di un oggetto e le forze che agiscono su di esso. Quando la precisione è troppo bassa gli errori derivati dagli arrotondamenti effettuati nei calcoli possono far sì che un oggetto oltrepassi o non raggiunga il punto esatto in cui dovrebbe trovarsi. Questi errori aumentano nelle situazioni in cui due oggetti liberi di muoversi sono vincolati tra loro con una precisione maggiore di quella che il motore fisico può calcolare. Questo può tradursi in un innaturale formazione di energia all'interno del sistema formato dai due oggetti che può manifestarsi all'inizio con un tremolio degli oggetti interessati fino ad arrivare ad un'esplosione delle due parti. Qualunque tipo di oggetto fisico composto e libero di muoversi può mostrare questo comportamento ma esso affligge con più facilità sistemi sottoposti ad alte tensioni interne e oggetti rotolanti. Una maggiore precisione

consente di ridurre gli errori di posizionamento e di applicazione delle forze interne al costo però di una maggiore potenza di calcolo richiesta.

Un altro inusuale aspetto della precisione dei motori fisici coinvolge il *framerate* o il numero di "momenti di tempo" per secondo in cui viene calcolata la fisica. Ciascun *frame* o momento viene trattato indipendentemente dagli altri e l'intervallo tra due *frame* non viene considerato. Un basso *framerate* e un piccolo oggetto che si sposti velocemente possono causare una situazione in cui l'oggetto non sembra muoversi dolcemente nello spazio, ma sembra tele-trasportarsi da un punto all'altro del suo percorso. A una velocità troppo alta un proiettile può mancare il bersaglio se questo è più piccolo del gap che intercorre tra i due momenti in cui viene calcolata la fisica (praticamente il proiettile appare prima e dopo il bersaglio e non si hanno collisioni benché la mira sia stata precisa!). In *Second Life* questo problema è risolto trattando ogni proiettile come se fosse una freccia. Una lunga e invisibile scia segue il proiettile in modo che possa coprire l'eventuale gap tra due frame consecutivi e permette così di rilevarne le collisioni.

La fisica su cui si basa l'animazione dei personaggi in passato è sempre stata trattata con la dinamica dei corpi rigidi per la velocità e la facilità di calcolo di questo sistema, tuttavia i moderni videogame e film stanno cominciando a utilizzare la fisica dei corpi morbidi quando possibile. Tale tecnica è utilizzata anche per gli effetti particellari come liquidi e vestiti. Infine alcune forme limitate di simulazione dinamica dei fluidi sono utilizzate talvolta per simulare acqua o altri liquidi oppure il fuoco e le esplosioni.

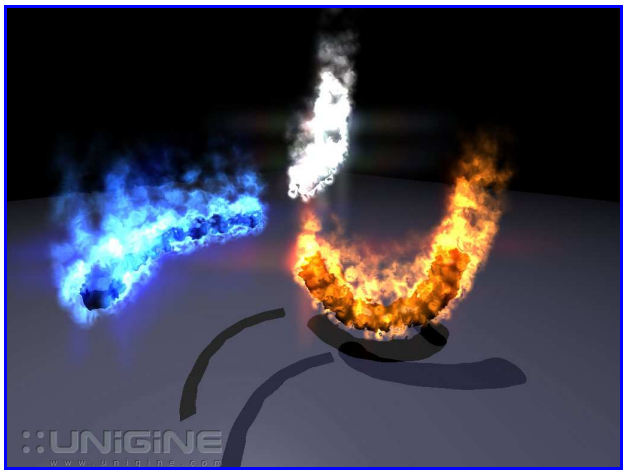

Figura 4 - Particle System

1.2 - Perché usare un Motore Fisico?

Forse ci si potrebbe chiedere perché sia necessario o comunque consigliabile ricorrere a un motore fisico già pronto e magari scritto da esperti anziché scrivere di propria mano il codice necessario a gestire la fisica delle nostre simulazioni. Per dare una risposta a questa domanda vediamo quali passi ci troveremmo a seguire se decidessimo di optare per la seconda ipotesi e scrivessimo da soli il nostro motore fisico. Proviamo a vedere come si dovrebbe ragionare ad esempio nel caso di una semplice simulazione del gioco del domino in cui abbiamo tante piccole tessere che interagiscono tra loro.

1.2.1 - Equazioni del moto della tessera

Partiamo descrivendo alcuni aspetti geometrici della nostra tessera che ci saranno utili in seguito.

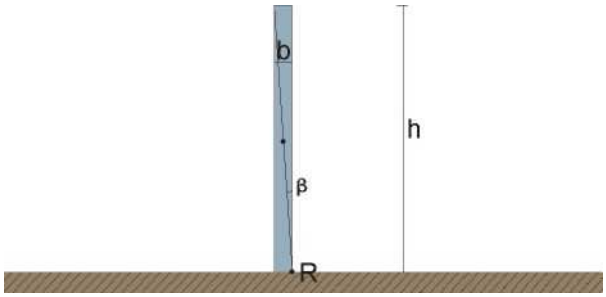

Figura 5 - Caratteristiche geometriche di una tessera del domino

Indichiamo con *h* l'altezza della tessera e con *b* la base; per il momento possiamo anche trascurare la profondità visto che il nostro moto si verifica solo nel piano *x*-*y*. Con β Indichiamo l'angolo formato dall'altezza e dalla diagonale della tessera; come noto dalla trigonometria questo vale:

$$\beta = \arctan(\frac{b}{h})$$

Un'altra grandezza importante per i nostri fini è la distanza tra il punto *R* e il baricentro G della tessera che è data da:

$$\overline{RG} = \frac{b}{2} \cdot \sin \beta$$

La caduta di una singola tessera del domino è riconducibile a un moto rotatorio attorno a un punto R della base della tessera. Il problema è che non è possibile scrivere un'unica equazione per la caduta, ma il moto va scomposto in diverse parti a seconda che ci sia o no il contatto con le tessere precedenti e/o seguenti. Cominciamo a vedere il moto della tessera soggetta alla forza *F* e alla propria forza peso *P* dalla posizione verticale fino a quando è ruotata di un angolo pari a β. Tale posizione rappresenta il limite entro il quale la forza peso agisce opponendosi alla rotazione; oltre tale angolo anche il peso della tessera contribuisce ad aumentare l'accelerazione angolare. Per semplicità inoltre supponiamo che F venga applicata alla tessera solo fino a questa posizione limite oltre la quale lasceremo cadere la tessera solo per effetto del proprio peso. Vediamo dunque il diagramma di corpo libero.

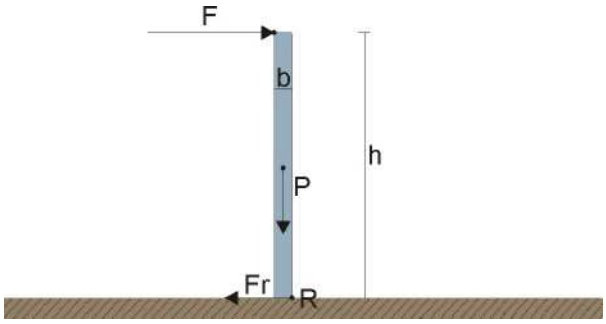

Figura 6 – Diagramma di corpo libero della tessera

P è la forza peso della tessera ed è data dalla nota relazione:

$$P = M \cdot g$$

Dove *M* è la massa della tessera e *g* è l'accelerazione di gravità.

F_r è la forza di attrito statico tra la tessera e il pavimento lungo la direzione orizzontale ed è di segno contrario rispetto a *F* in quanto essa, come ogni forza di attrito si oppone al movimento. Per definizione vale:

$$F_r = \mu \cdot F_n$$

Dove μ è il coefficiente di attrito statico e F_n è la forza normale esercitata dal pavimento sulla tessera. In questo caso F_n equivale a P.

F è la forza che applichiamo in cima alla tessera per produrre la caduta; tale forza deve essere minore di F_r per consentire che la tessera ruoti attorno al punto R senza scivolare, ma deve essere abbastanza grande in modo che la sommatoria del momento torcente da essa generata e del momento generato dalla forza-peso sia positiva.

Nel caso del moto rotazionale possiamo riscrivere la seconda legge della dinamica di Newton in questo modo:

$$\sum \tau = I \cdot \alpha$$

Questa formula ci dice che la sommatoria di tutti i momenti torcenti applicati alla tessera è uguale al prodotto tra il momento d'inerzia della tessera rispetto all'asse di rotazione e l'accelerazione angolare che questa subisce; In altre parole possiamo dire che un corpo che abbia un momento d'inerzia I e a cui applichiamo un momento τ comincerà a ruotare con un accelerazione angolare α. Nel caso della tessera del domino in figura 2 questi valgono:

$$\sum \tau = -M \cdot g \cdot \frac{b}{2} + F \cdot h$$

$$I = \frac{1}{3} \cdot M \cdot h^2$$

E ricordando che:

$$\alpha = \frac{d\omega}{dt} \quad \text{e} \quad \omega = \frac{d\vartheta}{dt}$$

Dove ϑ è l'angolo di cui ruota la tessera e ω è la velocità angolare. Possiamo così scrivere l'equazione del moto per la nostra tessera:

$$-M \cdot g \cdot \frac{b}{2} + F \cdot h = \frac{1}{3} \cdot M \cdot h^2 \cdot \frac{d\omega}{dt}$$

Se ora vogliamo ottenere un'equazione che descriva l'angolo di rotazione della tessera rispetto al punto *R* in funzione del tempo ci basta integrare due volte questa equazione rispetto alla variabile *tempo*.

L'equazione trovata presenta un'imprecisione: considera costanti i valori di *h* e *b* utilizzati come braccio delle forze. In realtà quando la tessera comincia a ruotare i due valori cominciano a variare e in particolare il braccio della forza peso passa da *b/2* a 0 e poi comincia a crescere fino a raggiungere il valore di *h/2* quando la tessera è caduta e si trova in posizione orizzontale, mentre per la forza *F* il dobbiamo fare delle considerazioni: se decidiamo di continuare ad applicare la forza perpendicolarmente alla nostra tessera, allora il braccio rimarrà costante, mentre se decidiamo di continuare ad applicarla lungo la direzione orizzontale il braccio passerà dal valore di *h* a 0. In questa simulazione si è optato però per una terza soluzione: La forza viene applicata ma solo per un breve periodo di tempo, sufficiente a generare un impulso in grado di produrre una variazione del momento angolare in grado di portare il baricentro della tessera appena oltre la verticale del punto *R*; a questo punto si può anche smettere di applicare la forza alla tessera in quanto basta la sola forza peso a generare un momento torcente positivo. Questa situazione limite è illustrata in figura 7.

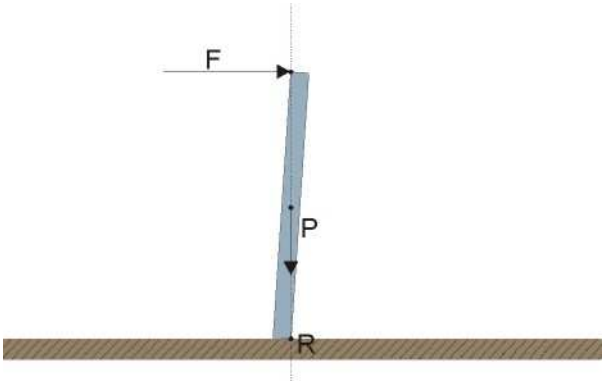

Figura 7 – Posizione limite in cui la forza peso si oppone alla rotazione

Proviamo ora a descrivere il moto di rotazione della tessera quando essa è soggetta solo alla propria forza peso. Questa situazione si verifica da quando smettiamo di applicare la forza *F* fino al momento in cui si ha l'urto con la tessera successiva. Osserviamo il diagramma in figura 4.

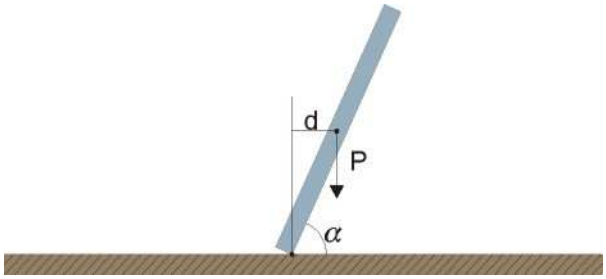

Figura 8 – La tessera in caduta libera

L'angolo α varia da 90° fino a 0° mentre l'angolo tra il baricentro della tessera e il centro della rotazione vale $\alpha+\beta$ pertanto possiamo dire che il braccio della forza peso vale:

$$d = \overline{RG} \cdot \cos(\alpha + \beta)$$

Pertanto, procedendo come prima avremo la nuova equazione del moto:

$$M \cdot g \cdot \cos(\alpha + \beta) = \frac{1}{3} M \cdot h^2 \cdot \frac{d\omega}{dt}$$

Analogamente a quanto fatto precedentemente, per trovare l'equazione che lega la rotazione al tempo dobbiamo risolvere l'equazione differenziale. A differenza di quanto fatto prima però, in questa equazione compare l'angolo α che è direttamente legato a θ, quindi in questo caso l'equazione differenziale è del secondo ordine.

Sempre procedendo in questo modo possiamo ricavare le equazioni del moto per le altre possibili configurazioni della caduta della tessera; in particolare è utile trovare le equazioni nel caso della tessera che spinge quella successiva (figura 5), della tessera che viene spinta dalla precedente (figura 6) e della tessera che si trova contemporaneamente a contatto con la tessera precedente e quella seguente (figura 7). In questi casi è possibili osservare la presenza di altre forze come quella esercitata da una tessera sull'altra o la forza di attrito tra due tessere che possono opporsi o favorire la rotazione. Naturalmente la presenza di queste forze va considerata nelle equazioni del moto complicando così i nostri calcoli.

Ora che abbiamo visto come ricavare le equazioni del moto della generica tessera durante le varie fasi della caduta ci resta da vedere cosa accade nel momento in cui due tessere collidono (figure 9, 10 e 11).

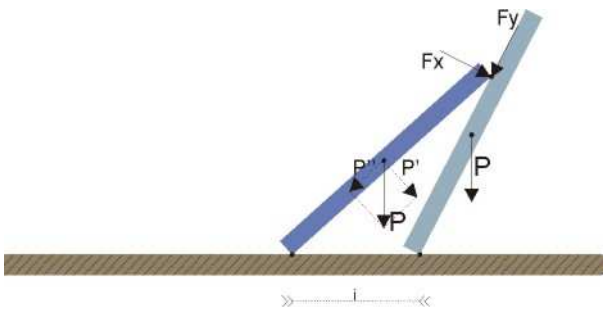

Figura 9 – Tessera spinta dalla precedente

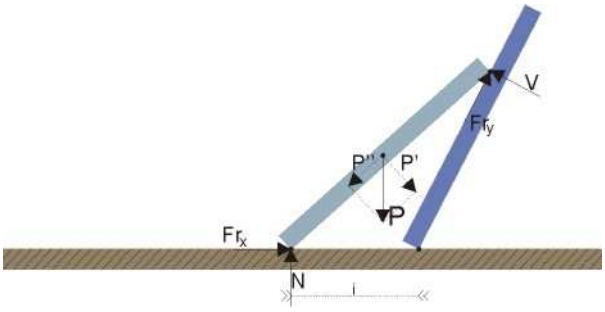

Figura 10 – Tessera che spinge la successiva

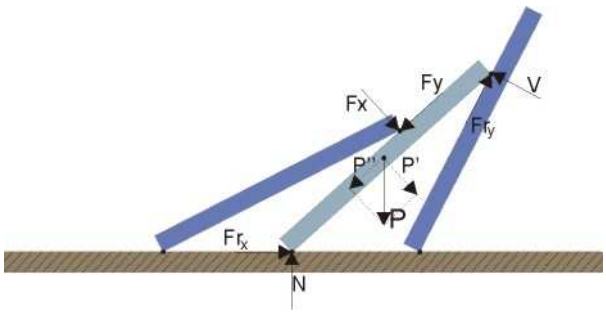

Figura 11 – Tessera in una generica posizione

1.2.2 - Urti tra due tessere

Quando la tessera tocca la successiva si verifica un urto. Durante un urto si hanno delle forze molto elevate che agiscono per brevi periodi di tempo. Inoltre le forze stesse non sono costanti ma crescono secondo un diagramma simile a quello mostrato in figura 12

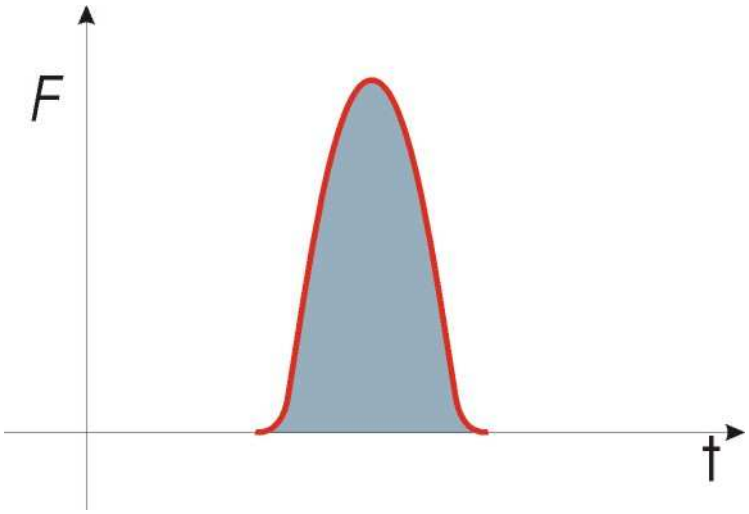

Figura 12 – L'andamento della forza durante un urto

Un altro fattore importante da tener presente quando si ha a che fare con gli urti sono le deformazioni che subiscono i corpi coinvolti. Proprio per questi motivi è spesso difficile trattare gli urti utilizzando la seconda di Newton mentre, sotto opportune condizioni, è preferibile un approccio basato sulle leggi di conservazione della quantità di moto e dell'energia cinetica.

La quantità di moto che è data da:

$$p = M \cdot v$$

Si conserva solo nell'ipotesi che sul sistema considerato non agiscano forze esterne. L'energia cinetica invece è:

$$KE = \frac{1}{2} M v^2$$

La conservazione dell'energia cinetica dipende dal tipo di urto. Distinguiamo così due tipi di urti: gli urti *elastici* in cui l'energia cinetica si conserva e gli urti *anelastici* in cui invece l'energia cinetica non si conserva. Dobbiamo anche dire che effettivamente non esistono urti totalmente elastici ma alcuni sistemi (come ad esempio l'urto di due biglie) li approssimano in modo piuttosto preciso. Ovviamente se anche l'energia cinetica non si conserva, l'energia totale del sistema si conserva e l'energia cinetica mancante sarà stata trasformata in altre forme (potenziale, termica, acustica, di deformazione). Se dopo l'urto i due corpi restano attaccati si ha un urto *completamente anelastico*.

Se nello studio di un urto possiamo applicare queste due leggi di conservazione allora possiamo anche determinare il comportamento dei corpi coinvolti dopo la collisione. Consideriamo ad esempio il caso di una palla da biliardo di massa *m* che si muove con velocità v_1 e collide frontalmente con una seconda palla ferma ($v_2=0$) e di ugual massa (figura 13).

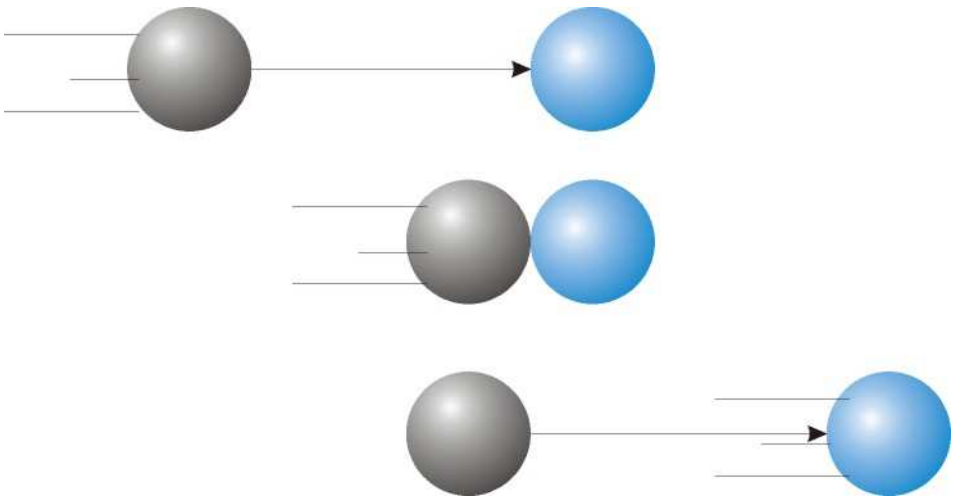

Figura 13 – Urto tra due palle da biliardo

Non essendoci forze esterne che agiscono sul sistema applichiamo la conservazione della quantità di moto:

$$m_1 v_1 + m_2 v_2 = m_1 v'_1 + m_2 v'_2 \quad \textbf{(a)}$$

E poiché supponiamo che le biglie siano rigide e che l'urto sia elastico applichiamo anche la legge di conservazione dell'energia cinetica.

$$\frac{1}{2}m_1v_1^2 + \frac{1}{2}m_2v_2^2 = \frac{1}{2}m_1v'^2_1 + \frac{1}{2}m_2v'^2_2 \qquad \textbf{(b)}$$

Riscriviamo ora le due equazioni raccogliendo a fattore comune le masse.

$$m_1(v_1 - v'_1) = m_2(v'_2 - v_2)$$
$$m_1(v_1^2 - v'^2_1) = m_2(v'^2_2 - v_2^2)$$

Se consideriamo che $(a-b)(a+b)=a^2-b^2$ possiamo anche scrivere l'equazione dell'energia cinetica come:

$$m_1(v_1 - v'_1)(v_1 + v'_1) = m_2(v'_2 - v_2)(v'_2 + v_2)$$

Se ora dividiamo questa equazione per l'equazione della quantità di moto otteniamo

$$v_1 + v'_1 = v'_2 + v_2$$

Che possiamo anche riscrivere come

$$v_1 - v_2 = -(v'_1 - v'_2) \quad \textbf{(c)}$$

Che ci dice che dopo l'urto la differenza di velocità tra le due biglie è uguale in modulo ma ha direzione opposta.

Ora possiamo riscrivere la **(a)** dividendo per la massa (uguale per entrambe le palle) e ponendo $v_1=v$ e $v_2=0$

$$v = v'_1 + v'_2$$

Siccome abbiamo due incognite ci serve una seconda equazione; utilizziamo allora la conservazione dell'energia cinetica nella forma **(c)**

$$v_1 - v_2 = v'_2 - v'_1 \qquad \textbf{(d)}$$

O

$$v = v'_2 - v'_1 \qquad \textbf{(e)}$$

Sottraendo la **(e)** dalla **(d)** otteniamo

$$0 = 2v'_1$$

Che ci dice che la velocità della prima palla dopo l'urto è uguale a 0. inserendo il risultato trovato nella equazione della quantità di moto otteniamo anche la velocità della seconda palla dopo l'urto

$$v'_2 = v$$

In pratica abbiamo visto che dopo l'urto elastico frontale le due palle si sono scambiate le relative velocità.

Abbiamo visto come utilizzare le leggi di conservazione della quantità di moto e dell'energia cinetica per studiare gli urti elastici tra due palle da biliardo. Purtroppo nel caso del domino queste leggi non valgono per due motivi: il primo è che l'urto tra due tessere è in generale anelastico visto che esse rimangono attaccate dopo l'urto e quindi non c'è conservazione dell'energia cinetica, il secondo è che il nostro sistema formato dalle tessere è soggetto anche a forze esterne (la gravità, che provoca un momento torcente che varia in funzione dell'angolo di rotazione della tessera) che ci impediscono di utilizzare anche la legge di conservazione della quantità di moto.

Queste difficoltà, unite alle difficoltà di risolvere le complesse equazioni del moto delle tessere dovrebbero rendere più chiaro il motivo per cui è preferibile utilizzare un motore fisico già pronto anziché crearne uno nuovo partendo da zero.

1.3 - Criteri di scelta e problemi tipici

I criteri per la selezione del motore fisico, viste le differenti tipologie di prodotti disponibili, saranno stabiliti principalmente sulla base del contesto applicativo e possono coinvolgere vari altri aspetti, che vanno dall'eventuale costo delle licenze alla visibilità e modificabilità del codice del motore.

Al fine di gestire razionalmente la procedura di selezione abbiamo adottato un *framework* di acquisizione software nato nell'ambito della produzione di software Open Source, denominato BRR, che sarà esposto in dettaglio nel capitolo tre.

2 - I motori fisici scelti

Vediamo ora di introdurre i due motori fisici che abbiamo scelto come candidati principali per le nostre simulazioni.

2.1 - Open Dynamics Engine (ODE)

2.1.1 - Cos'è ODE e cosa non è

Open Dynamics Engine è una libreria software Open Source per la modellazione basata sulla fisica di corpi rigidi e articolati. Le sue migliori caratteristiche sono la velocità, la flessibilità e la robustezza. Inoltre ODE dispone di un sistema di rilevamento delle collisioni integrato. ODE è stato inizialmente sviluppato da Russell Smith e il sito di riferimento è *www.ode.org.*

ODE non è una libreria grafica, non è un *framework* per la gestione di oggetti deformabili e non è una libreria per la simulazione di sistemi particellari.

Tra le sue caratteristiche spiccano la gestione dell'attrito, la possibilità di implementare un proprio sistema di *collision-detection* e la facilità d'uso e di integrazione con altre librerie grafiche.

2.1.2 - Preparazione alla simulazione

Prima di cominciare la simulazione vera e propria sono necessari alcuni passi preliminari:

- Creazione di un *dynamics world*
- Creazione dei corpi rigidi ed inserzione nel *world*
- Impostazione dei parametri dei corpi rigidi: posizione, orientamento ecc.
- Creazione di eventuali vincoli meccanici
- Collegamento dei vincoli ai corpi interessati
- Impostazione dei parametri dei vincoli
- Creazione di un *collision world*

Vediamo ora di analizzare un po' più in dettaglio queste operazioni per vedere più da vicino alcune delle funzionalità offerte dalla libreria.

2.1.3 - Creazione di un dynamics world

Nelle simulazioni ODE un *dynamics world* è una struttura destinata a contenere i corpi rigidi che parteciperanno alla simulazione vera e propria . Un *dynamics world* viene creato con la seguente istruzione:

```
dWorldID w = dWorldCreate();
```

Se ora desideriamo applicare a questo mondo la forza di gravità terrestre ci basta usare la funzione:

```
dWorldSetGravity(w,0,0,-9.1);
```

Un'altra importante funzione è quella che permette l'evoluzione di un *dynamics world* che qui citiamo ma che riprenderemo meglio in seguito:

```
dWorldStep(w,0.01);
```

2.1.4 - Creazione dei corpi rigidi ed inserzione nel world

I corpi rigidi da inserire nel world vengono creati con il comando

```
dBodyID b = dBodyCreate(w);
```

Come si può vedere il parametro passato alla funzione indica in quale world deve essere inserito il corpo appena creato.

2.1.5 - Impostazione delle caratteristiche dei corpi rigidi

Una volta creati i corpi rigidi se ne possono settare le caratteristiche principali che dovranno avere al momento in cui comincia la simulazione. Vediamo quali sono alcune di queste caratteristiche e segnaliamo che per ogni ognuna di queste funzioni esiste una rispettiva funzione "get" che permette di leggere la particolare caratteristica.

```
dBodySetPosition(b,x,y,z); //imposta le coordinate xyz del
corpo b

 dBodySetRotation(b,R); //imposta la rotazione del corpo b
usando    la matrice di rotazione R
```

```
dBodySetQuaternion(b,q); //imposta la rotazione di b usando
il quaternione q

dBodySetLinearVel(b,x,y,z); // setta la velocità lineare del
corpo b con il vettore di componenti xyz

dBodySetAngularVel(b,x,y,z); // setta la velocità angolare
del corpo con il vettore di componenti xyz
```

Una caratteristica importante dei corpi rigidi è la massa. In ODE non si setta semplicemente la massa del corpo ma bisogna anche indicare la distribuzione di massa attorno al baricentro del corpo. Questa distribuzione di massa viene rappresentata da una matrice di inerzia di cui si può settare manualmente ogni valore; è tuttavia molto più semplice utilizzare delle funzioni che semplificano la creazione di distribuzioni di massa tipiche di alcune basilari primitive geometriche. Ad es:

```
dMass m;  //crea una variabile di tipo dMass vuota

dMassSetSphere(&m,1,0.5); //riempie i campi della struttura
dMass con i valori che descrivono una sfera di raggio 0.5 e
densità 1

dBodySetMass(b,&m); //assegna al corpo b la massa m
```

Esistono poi numerose altre funzioni in grado di combinare le primitive geometriche per ottenere distribuzioni di massa in grado di descrivere forme più complesse.

2.1.6 - Creazione dei vincoli

Un vincolo è una relazione che viene stabilita tra due corpi in modo che essi possano assumere solo determinate posizioni/orientamenti l'uno rispetto all'altro. I vincoli rimuovono gradi di libertà dai possibili movimenti dei corpi rigidi. Ogni tipo di vincolo ha una serie di parametri che permettono di realizzare sistemi meccanici di vario tipo. Ecco alcune funzioni che permettono di creare dei particolari vincoli:

```
dJointID j = dJointCreateBall(w,0); //crea un vincolo di tipo
ball

dJointSetBallAnchor (j,x,y,z); // posiziona l'anchor del
vincolo di tipo ball
```

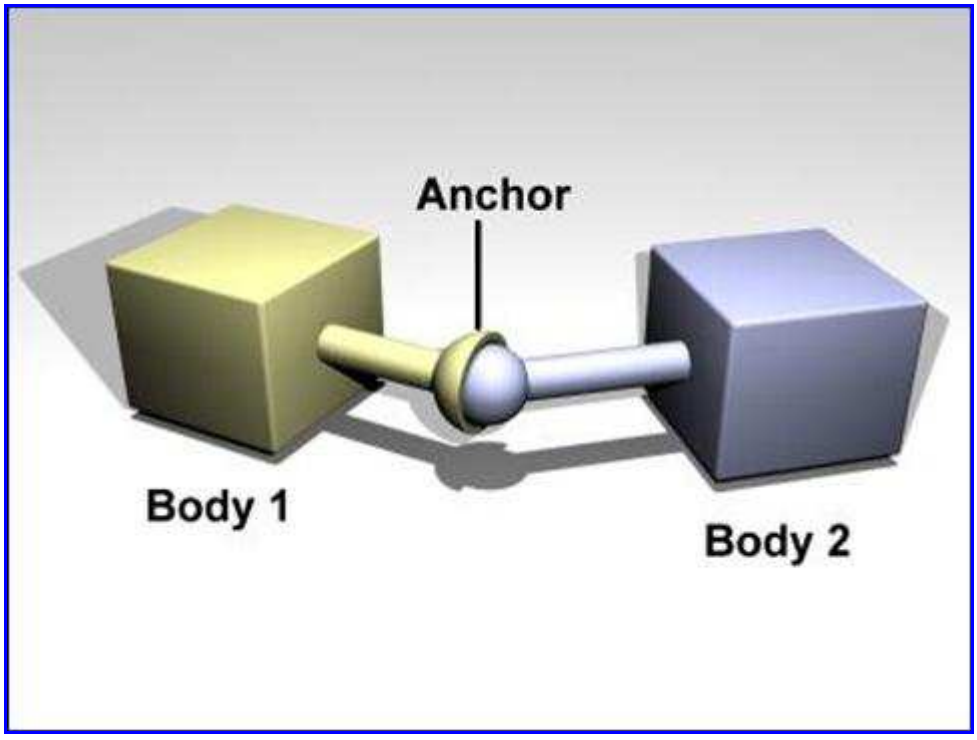

Figura 14 - Vincolo di tipo ball

```
dJointID j = dJointCreateHinge(w,0); //crea un vincolo di
tipo Hinge

dJointSetHingeAnchor(j,x,y,z); // setta la posizione
dell'anchor

dJointSetHingeAxis(j,x,j,z);//setta l'orientamento dell'asse
```

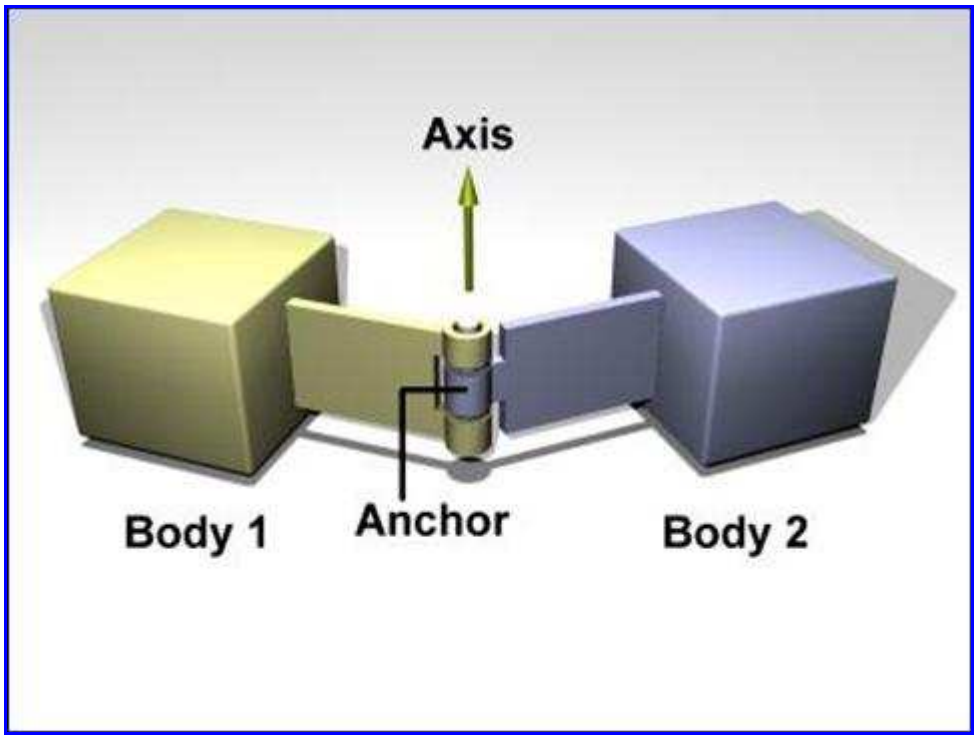

Figura 15 – Vincolo di tipo hinge

```
dJointID j = dJointCreateSlider( w, 0 );// crea un vincolo di
tipo slider

dJointSetSliderAxis( j, x, y, z ); //setta l'orientamento
dell'asse
```

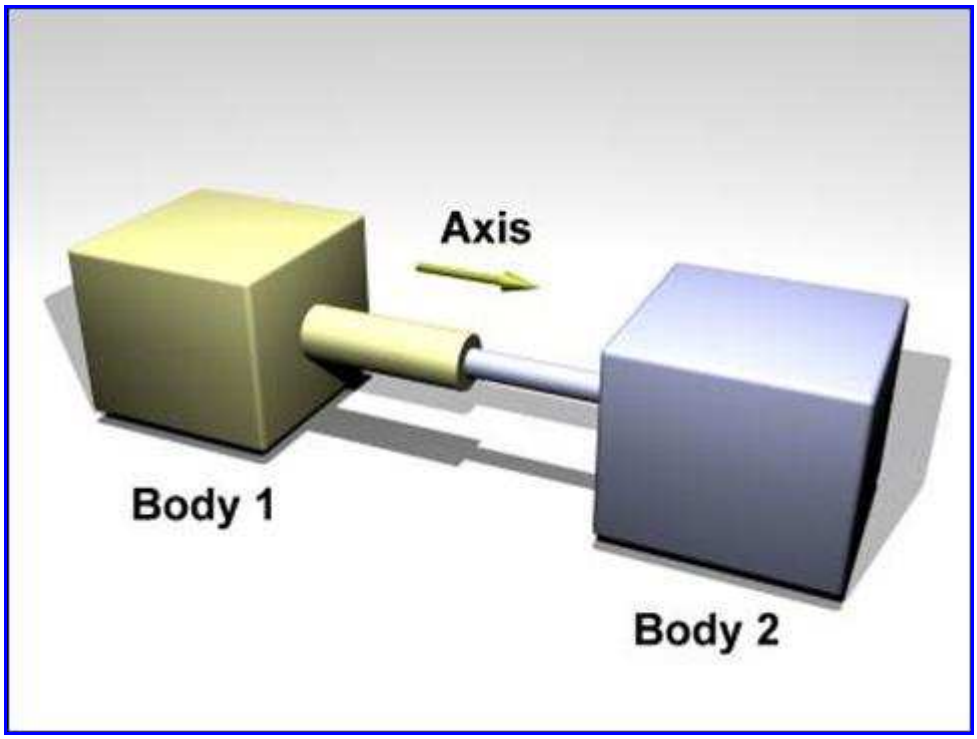

Figura 16 - Vincolo di tipo slider

```
dJointID j = dJointCreateUniversal( w, 0 );// crea un vincolo
universale

dJointSetUniversalAnchor( j, x, y, z); //setta la posizione
di ancoraggio

dJointSetUniversalAxis1( j, x, y, z); //setta l'orientamento
dell'asse 1

dJointSetUniversalAxis2( j, x, y, z); //setta l'orientamento
dell'asse 2
```

Questi sono alcuni dei tipi di vincoli che è possibile creare in ODE. Per ognuno di questi ci sono diversi parametri che è possibile configurare per ottenere un'ampia gamma di differenti situazioni che è possibile simulare. Due parametri in particolare vanno citati: CFM ed ERP. CFM sta per *Constraints Force Mixing* e, detto in parole povere, è un valore che indica quanto è possibile violare un vincolo. Sebbene il fatto di violare un vincolo possa

apparire strano, questo permette di ricreare situazioni come le molle o i materiali più soffici. ERP sta per *Error Reduction Parameter*; dopo ogni step di simulazione a causa dell'imprecisione dell'integratore usato si creano degli errori nel posizionamento dei corpi rigidi e dei vincoli. ODE provvede quindi, mediante un meccanismo interno, a ridurre questi errori: ERP indica quanto vanno ridotti gli errori. Ovviamente valori più alti di ERP danno luogo a una maggior precisione ma peggiorano le prestazioni. CFM e ERP possono essere usati anche per definire le medesime proprietà per l'intera simulazione.

Un ultimo particolare tipo di vincolo che vale la pena di considerare è il vincolo di contatto. Esso viene creato quando due corpi collidono. Una volta creato se ne possono settare i parametri come per qualunque altro tipo di vincolo. Si vedrà meglio l'utilizzo di questo tipo di vincolo nel capitolo dedicato alle collisioni.

```
dJointID c = dJointCreateContact(w,cntgrp,&cnt[i]); //crea un
vincolo di contatto
```

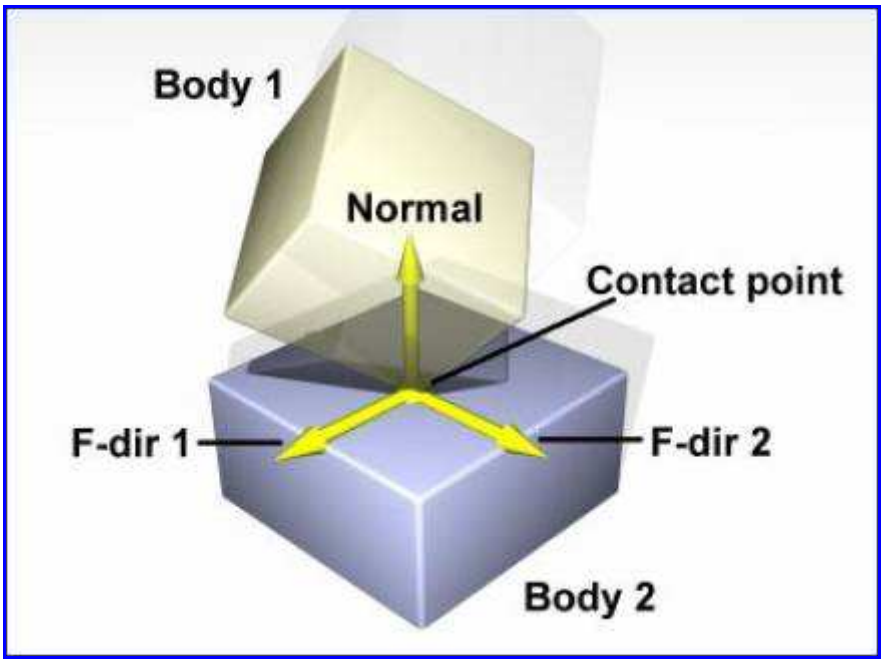

Figura 17 – Vincolo di contatto

Naturalmente, come esistono funzioni per settare i vari parametri , esistono anche funzioni per ricavare questi valori da uno specifico vincolo.

2.1.7 - Collegamento dei corpi rigidi ai vincoli

Una volta creati i vincoli e i corpi rigidi non resta altro da fare che collegarli tra loro. Per fare questo è sufficiente usare la funzione:

```
dJointAttach(j, b1, b2); //unisce b1 e b2 per mezzo del
vincolo j
```

Utilizzando queste semplici funzionalità è possibile arrivare a creare strutture anche molto complesse che simulano il comportamento di oggetti reali (vedi fig. 18)

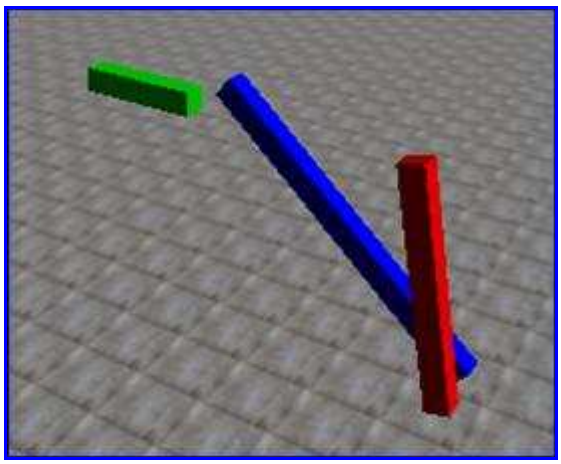

Figura 18 – Sistema biella-manovella in 3D

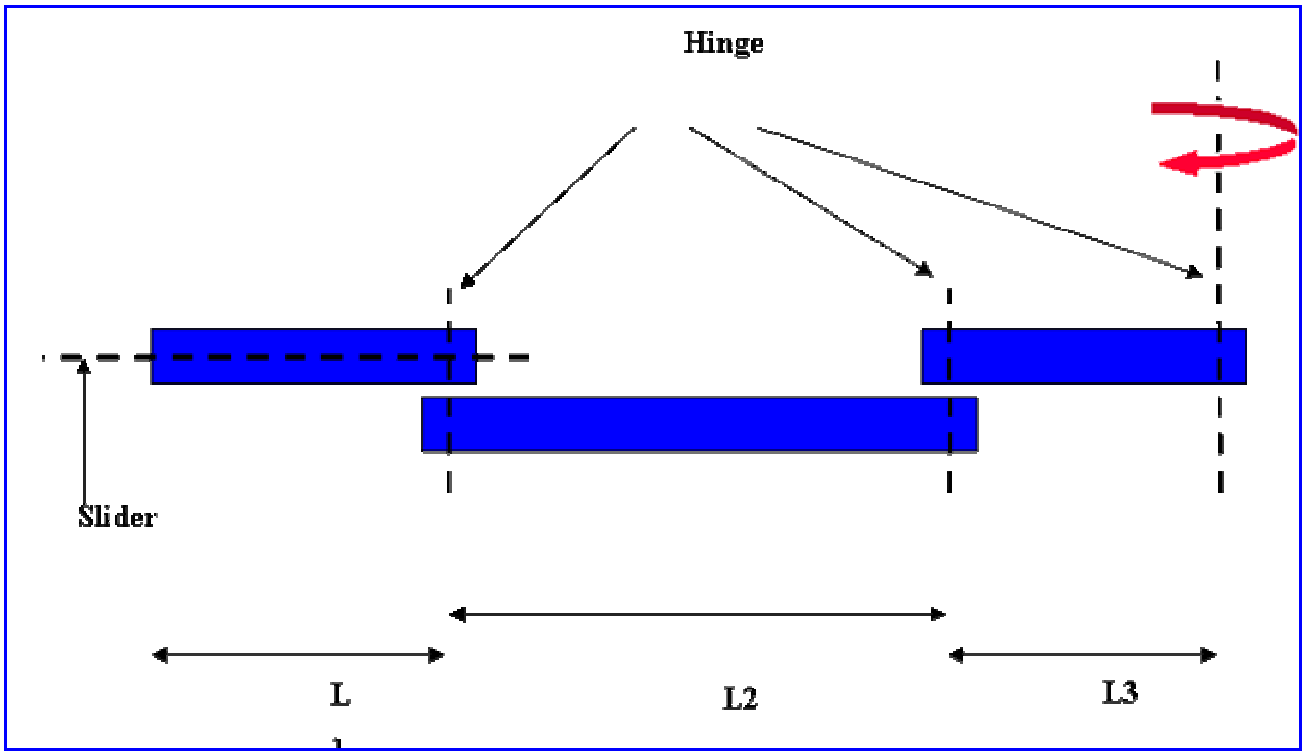

Figura 19 – schema di sistema biella manovella costruito con i vincoli di ODE

2.1.8 - Impostazione dei parametri dei vincoli

I parametri che si possono settare dipendono dal tipo di vincolo in questione. In generale questi includono le posizioni dei punti di ancoraggio, gli assi di rotazione, i coefficienti di attrito, i coefficienti di restituzione usati nelle collisioni, i valori di CFM ed ERP. È da sottolineare che quando due corpi vengono collegati a un vincolo viene valutata la loro posizione rispetto al vincolo e tale posizione viene considerata come posizione zero; in base a questa posizione vengono decisi altri parametri come il massimo angolo che possono formare i due corpi rispetto al vincolo o la massima e la minima distanza a cui si possono trovare.

2.1.9 - Creazione del collision world

Tutti i passaggi visti finora sono serviti a creare il mondo dinamico di ODE. Nonostante, come si è visto, si sia potuta settare la distribuzione di massa del corpo per definire delle forme solide, in questo punto della simulazione i corpi rigidi sono considerati come puntiformi. In un mondo fatto di oggetti puntiformi è improponibile cercare di simulare la realtà. Nella realtà infatti bisogna anche tenere conto delle collisioni tra i vari oggetti e una collisione è diversa a seconda del fatto che due corpi si urtino in pieno oppure solo di striscio. Serve dunque un modo per definire le forme dei vari corpi e per valutare come si muovono queste forme nel tempo e nello spazio al fine di valutare correttamente le interazioni tra i corpi. Per venire incontro a questa esigenza ODE propone il *collision world* che è appunto uno strumento che serve a definire e valutare le posizioni e le interazioni dei corpi rigidi. Per cominciare si crea uno spazio delle collisioni con la funzione:

```
dSpaceID s = dSimpleSpaceCreate(0);
```

In alternativa è possibile creare uno spazio basato su una tabella *hash* che permette di migliorare le prestazioni quando nella simulazione sono presenti molti oggetti. Uno spazio può contenere forme geometriche semplici, complesse o addirittura altri spazi (per migliorare le prestazioni). Una volta creato lo spazio dunque lo si può riempire con delle geometrie. ODE mette a disposizione alcune primitive per la creazione di forme geometriche:

```
dGeomID g = dCreateSphere(s,  r); //crea una sfera di raggio
r

dGeomID g = dCreateBox (s, x, y, z); //crea un
parallelepipedo di dimensioni x y z
```

```
dGeomID g = dCreatePlane(s, x, y, z); //crea un piano la cui
normale è il vettore x y z

dGeomiD g = dCreateCCylinder(s, r, h); //crea un cilindro di
raggio r e altezza h

dGeomID g = dCreateRay(s, l);//crea un raggio di lunghezza l

dTriMeshDataID m = dGeomTriDataMeshCreate(); //crea una mesh
triangolare
```

In particolare le ultime due funzioni hanno utilizzi particolari: un raggio è spesso usato per verificare la visibilità di un oggetto oppure la traiettoria di un corpo rigido. La funzione che crea la *mesh* invece serve solo per inizializzare la struttura dati che poi andrà riempita con i vertici della *mesh* stessa. Collegando tra loro più *mesh* è possibile realizzare forme anche molto complesse. Ogni geometria creata può essere modificata in seguito con funzioni analoghe a quelle utilizzate per i corpi rigidi per variarne dimensioni, posizione e rotazione, ma è molto più semplice legare tra loro un corpo rigido e la rispettiva forma in modo che qualunque modifica eseguita sul corpo si ripercuota sulla sua geometria:

```
dGeomSetBody (g, b); //Lega la forma g al corpo rigido b
```

Ora che anche il *collision world* è stato preparato siamo pronti per la simulazione vera e propria.

2.1.10 - La simulazione

La simulazione ODE è costituita principalmente da quattro passi:

- Applicazione delle forze e dei momenti ai corpi rigidi
- Regolazione dei parametri dei vincoli
- Determinazione delle collisioni
- Esecuzione di un passo di simulazione

Vediamo in dettaglio queste fasi.

2.1.11 - Applicazione delle forze e dei momenti ai corpi rigidi

In questa fase si applicano forze e momenti che determineranno il comportamento del sistema fisico. Segnaliamo le principali funzioni utilizzate.

```
dBodyAddForce(b, fx, fy, fz);//applica al corpo la forza
rappresentata dal vettore fx fy fz

dBodyAddTorque(b, fx, fy, fz);//aggiunge al corpo un momento
torcente

// queste funzioni permettono di specificare anche il punto
di applicazione delle forze e dei momenti

dBodyAddRelForce(b, fx, fy, fz);

dBodyAddRelTorque(b, fx, fy, fz);

dBodyAddForceAtPos( b, fx, fy, fz, px, py, pz);

dBodyAddForceAtRelPos( b, fx, fy, fz, px, py, pz);

dBodyAddRelForceAtPos( b, fx, fy, fz, px, py, pz);

dBodyAddRelForceAtRelPos( b, fx, fy, fz, px, py, pz);
```

2.1.12 - Impostazioni dei parametri dei vincoli

In questa fase si possono regolare i parametri dei vincoli del sistema. La maggior parte di essi è comunque stata regolata durante la creazione, quindi in questa fase solitamente ci si limita a cambiare i valori di CFM ed ERP come già spiegato in precedenza per simulare particolari comportamenti.

2.1.13 - Determinazione delle collisioni

La determinazione delle collisioni (*collision detection*) è a sua volta composta da altre tre sotto-fasi:

- Creazione di una lista di punti di contatto
- Creazione di un vincolo di contatto
- Impostazione delle proprietà del contatto: attrito, elasticità ecc.

Un punto di contatto è una particolare struttura che contiene informazioni come: la posizione del punto di contatto, la normale al punto, la profondità di penetrazione, le due geometrie coinvolte. Per determinare se due geometrie collidono e quali sono i punti di contatto esistono due vie. La prima utilizza la funzione:

```
dCollide(o1, o2, max_contacts, contact_array, skip);
```

Questa funzione prende due geometrie e verifica se collidono riempiendo *contact_array* con al massimo *max_contacts* punti di contatto. Applicando questo metodo per ogni possibile coppia di geometrie si possono così rilevare tutte le collisioni. Tale strategia è, algoritmicamente parlando, dell'ordine di $O(n^2)$ cioè cresce col quadrato del numero di oggetti. Per migliorare le prestazioni si può utilizzare la funzione:

```
dSpaceCollide(s, data, &nearCallback);
```

In questo modo il motore fisico si fa carico di trovare solo le coppie di geometrie potenzialmente a rischio di collisione e su queste chiama una funzione di *callback* che poi gestisce effettivamente la collisione spesso utilizzando ancora la precedente dCollide() .

Trovati tutti i punti di contatto si crea per ognuno di essi un vincolo di contatto di cui poi si setteranno tutti i parametri voluti per il comportamento che si vuole ottenere. Alcuni parametri che è possibile regolare sono:

- Attrito lungo la direzione principale
- Attrito lungo la direzione ortogonale alla principale
- Coefficiente di restituzione (elasticità dell'urto)
- CFM
- ERP

2.1.14 - Esecuzione di un passo di simulazione

Dopo tutti questi passi è giunta l'ora di far evolvere il nostro sistema. Quando si esegue questo passo ODE riceve in ingresso un intervallo temporale e, sulla base di tutte le informazioni ricevute, calcola come cambia il sistema dopo tale intervallo di tempo. Per eseguire un passo di simulazione si può usare una delle tre seguenti funzioni:

```
dWorldStep(w, t); //evolve il mondo w di un periodo di tempo t

dWorldStepFast1(w, t, n);//evolve il mondo w di un tempo t usando per l'algoritmo n iterazioni

dWorldQuickStep(w, t, n);//evolve il mondo w di un tempo t usando per l'algoritmo n iterazioni
```

La prima funzione è quella più precisa ma che utilizza più risorse e rallenta il sistema. Le altre due sono simili ed utilizzano un algoritmo iterativo per calcolare l'evoluzione del sistema; con queste due funzioni è possibile indicare il numero di iterazioni

che si vogliono eseguire. Ovviamente maggiore è il numero di iterazioni e maggiore sarà la precisione dei calcoli.

ODE dispone inoltre di una insieme di caratteristiche che si possono attivare o meno per aumentare la velocità dei calcoli senza diminuire la precisione; tra queste la più importante che vale la pena di citare è la possibilità di escludere automaticamente dai calcoli i corpi che sono rimasti inattivi per un certo periodo di tempo, salvo poi riattivarli automaticamente.

2.2 - Newton game dynamics

2.2.1 - Cos'è Newton Game Dynamics

Newton Game Dynamics è una soluzione integrata per la simulazione in tempo reale di ambienti fisici. Le API del software forniscono il controllo della scena, il rilevamento delle collisioni e il comportamento dinamico; *Newton Game Dynamics* è un software piccolo, veloce, stabile e facile da utilizzare.

Il motore implementa un risolutore deterministico che non è basato sui tradizionali metodi LCP (*Linear Complementarity Problem*) e iterativi ma che possiede rispettivamente la stabilità e la velocità di entrambi. Questa caratteristica rende il prodotto uno strumento adatto non solo per i videogame, ma anche per qualunque simulazione fisica in tempo reale.

È possibile integrare *Newton Game Dynamics* nei propri progetti con facilità. La tecnologia su cui si basa fa sì che l'utente debba conoscere solo i principi fisici di base per riprodurre semplicemente comportamenti fisici realistici. Il tempo di setup è ridotto al minimo perché non c'è bisogno di conoscere a priori l'hardware su cui girerà il software così come non c'è bisogno di settare strani parametri per avere un timing corretto nelle simulazioni. Il motore fisico di *Newton Game Dynamics* si comporta in modo molto naturale.

2.2.2 - Preparare una simulazione con Newton Game Dynamics

La preparazione di una simulazione fisica con *Newton Game Dynamics* richiede alcuni passaggi obbligatori e altri facoltativi, vediamo quali sono quelli più importanti:

- Creazione di un *Newton world*
- Dimensionamento del *Newton world*
- Scelta del risolutore
- Scelta del modello di attrito
- Creazione delle geometrie
- Creazione dei corpi rigidi
- Creazione dei vincoli tra i corpi

2.2.3 - Creazione di un Newton world

Il *Newton world* è la struttura dati principali del software. Esso rappresenta l'universo fisico entro cui si svolge la simulazione ed è destinato a contenere gli oggetti che vogliamo simulare. La creazione del *Newton world* avviene tramite la funzione:

```
NewtonWorld* NewtonCreate(NewtonAllocMemory mallocFnt,
NewtonFreeMemory mfreeFnt)
```

NewtonAllocMemory mallocFnt è un puntatore a una funzione di *callback* per l'allocazione della memoria definite dall'utente. Se il parametro è NULL viene usata la funzione standard.

NewtonFreeMemory mfreeFnt è un puntatore a una funzione di *callback* per il rilascio della memoria definita dall'utente. Anche in questo caso se il parametro è NULL viene utilizzata la funzione standard.

Vi sono poi altre funzioni che permettono di gestire il *Newton world*, tra queste le più importanti che meritano di essere citate sono:

```
void NewtonDestroy(const NewtonWorld* newtonWorld)
```

Distrugge il *Newton world* che gli viene passato come parametro

```
dFloat NewtonGetGlobalScale(const NewtonWorld* newtonWorld)
```

Restituisce il fattore di scala del *Newton world*. Il sistema fisico in teoria non dovrebbe essere legato alle dimensioni, tuttavia in pratica il motore deve implementare con una precisione limitata i numeri in virgola mobile e pertanto è inevitabile ricorrere a fattori di tolleranza per aumentare le prestazioni e la stabilità. Questi fattori di tolleranza rendono il motore fisico dipendente dalle dimensioni, per esempio diciamo che un corpo rigido viene considerato fermo quando la sua velocità è minore 0,01 unità per secondo per un certo numero di frames consecutivi. Per un programma che usa i metri come unità di misura questo si traduce in 0,01 metri al secondo, tuttavia per un programma che utilizza i centimetri come unità di misura significa 0,0001 metri al secondo, il che si traduce con una maggiore difficoltà del motore fisico a "congelare" il corpo nel secondo caso con un evidente incremento della capacità di calcolo richiesta. Una soluzione a questo problema consiste nello scalare tutte le tolleranze in modo da adattarle alle unità in uso nel programma.

```
void NewtonSetPlatformArchitecture(const NewtonWorld*
newtonWorld, int mode)
```

Questa funzione permette all'applicazione di configurare *Newton Game Dynamics* in modo che possa trarre vantaggio dall'architettura hardware sottostante. Il parametro mode identifica il metodo utilizzato: 0 è l'unico valore il cui funzionamento è garantito su tutte le piattaforme, con il valore 1 *Newton Game Dynamics* tenta di utilizzare specifiche istruzioni della CPU per la gestione dei numeri in virgola mobile, con il valore 2 il motore fisico cerca di utilizzare tutte le possibili istruzioni hardware che permettono di ottenere un incremento di prestazioni; questo è anche il valore di default.

```
void NewtonSetMinimumFrameRate(const NewtonWorld*
newtonWorld, dFloat frameRate)
```

Setta il minimo framerate a cui deve girare la simulazione. Per default il minimo valore di frame rate è impostato a 60 frame per secondo. Quando la simulazione scende al di sotto di tale soglia *Newton Game Dynamics* ottimizza i passaggi per il calcolo degli oggetti in modo da soddisfare le richieste minime.

```
void NewtonSetBodyLeaveWorldEvent(const NewtonWorld*
newtonWorld, NewtonBodyLeaveWorld callback)
```

Permette di definire una funzione di *callback* che servirà per decidere il comportamento dei corpi che finiscono al di fuori del mondo fisico.

2.2.4 - Dimensionamento del Newton world

Il *Newton world* deve avere una dimensione finita. Le dimensioni del mondo per default vengono impostate a quelle di una scatola di ±100 unità in tutte e tre le dimensioni quando viene creato il mondo e quando viene invocata la funzione *NewtonRemoveAllBodies()*. Per variare queste dimensioni è sufficiente utilizzare la funzione:

```
void NewtonSetWorldSize(const NewtonWorld* newtonWorld, const
dFloat* minPtr, const dFloat* maxPtr)
```

minPtr è un puntatore a un array che contiene le coordinate spaziali dell'estremo inferiore del *Newton world*; **maxPtr** è un puntatore a un array che contiene le coordinate spaziali dell'estremo superiore del *Newton world*.

2.2.5 - Scelta del risolutore

Newton Game Dynamics permette di scegliere il tipo di risolutore che si desidera utilizzare in funzione delle esigenze dell'utente. Per farlo si utilizza la funzione:

```
void NewtonSetSolverModel(const NewtonWorld* newtonWorld, int mode)
```

Il valore 0 rappresenta il metodo esatto; questo metodo è utile per applicazioni dove la precisione è più importante della velocità. Il valore 1 indica il metodo adattivo: il risolutore non è esatto ma la simulazione mantiene comunque un alto grado di accuratezza; è indicato per applicazioni in cui la precisione è importante ma non quanto la velocità. Con un generico valore n si identifica il metodo lineare; in questa modalità il risolutore non prova a ridurre gli errori di accelerazione nei vincoli sotto un certo limite, applica invece n passaggi sulla configurazione dei vincoli, cercando a ogni passaggio di ridurre gli errori, ma termina una volta eseguiti gli n passaggi indipendentemente dal grado di accuratezza raggiunto. In generale questo è il metodo più veloce ed è utilizzato in applicazioni in cui la velocità è il fattore preponderante, il classico esempio è quello dei videogame.

2.2.6 - Scelta del modello di attrito

È possibile decidere il grado di accuratezza con cui si intende riprodurre le forze di attrito. La funzione utilizzata a questo scopo è la seguente:

```
void NewtonSetFrictionModel(const NewtonWorld* newtonWorld, int model)
```

I valori ammessi sono solo 0 e 1. Con 0 si indica il modello esatto. Le forze di attrito sono calcolate in ciascun frame; questo modello è utile in simulazioni in cui è importante il grado di precisione. Il valore 1 rappresenta il modello adattivo; in questo modello i valori di attrito dei frame precedenti sono utilizzati per calcolare il massimo attrito del frame corrente; questo metodo è più veloce di circa il 10% rispetto al modello esatto ma può introdurre strani comportamenti delle forze di attrito: per esempio un oggetto rimbalzante che sta cadendo per una rampa si comporta come un oggetto senza attrito in quanto il contatto tra rampa e oggetto non è continuo.

2.2.7 - Creazione delle geometrie

Un aspetto fondamentale degli oggetti presenti in un sistema fisico è la geometria che descrive gli oggetti stessi. Essa è importante per descrivere la distribuzione della massa dell'oggetto nonché per identificare le collisioni tra gli oggetti stessi. *Newton Game Dynamics* mette a disposizione diverse primitive per costruire geometrie semplici e complesse; eccone alcune tra le principali:

```
NewtonCollision* NewtonCreateNull(const NewtonWorld*
newtonWorld)
```

Crea una primitiva di collisione trasparente. Può essere utile quando si desiderano avere dei corpi rigidi che non collidono mai.

```
NewtonCollision* NewtonCreateBox(const NewtonWorld*
newtonWorld, dFloat dx, dFloat dy, dFloat dz, const dFloat
*offsetMatrix)
```

Crea una primitiva a forma di parallelepipedo; *dx*, *dy*, *dz* sono le dimensioni del parallelepipedo lungo i tre assi principali mentre *offsetMatrix*, se presente, indica lo scostamento della geometria dal centro di massa del corpo rigido.

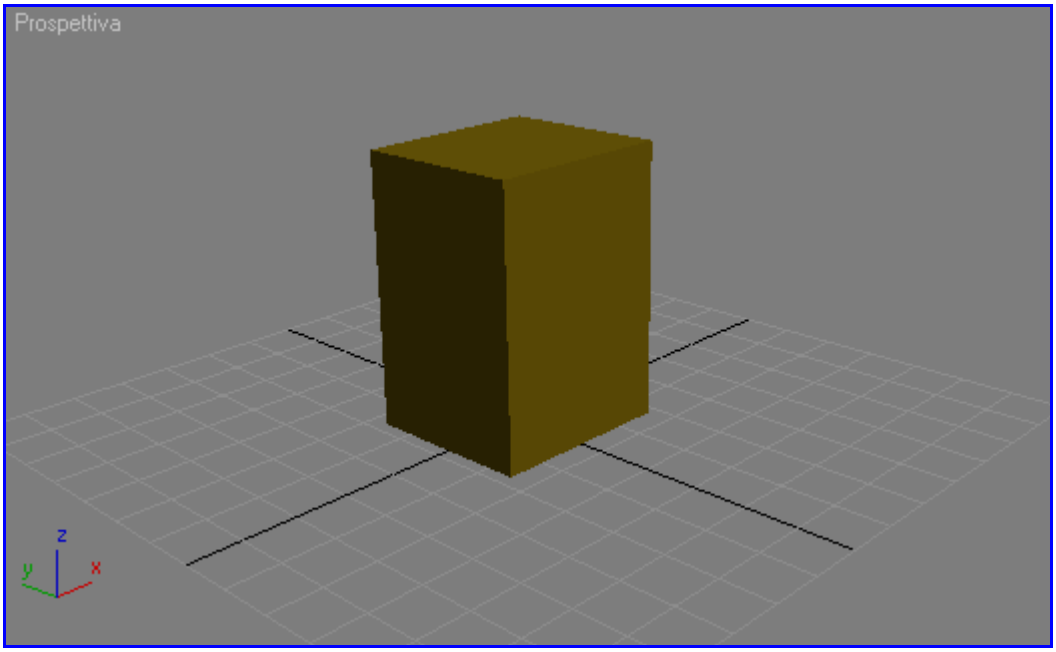

Figura 20 – La primitiva box

```
NewtonCollision* NewtonCreateSphere(const NewtonWorld*
newtonWorld, dFloat radiusX, dFloat radiusY, dFloat radiusZ,
const dFloat *offsetMatrix)
```

Crea una primitiva geometrica a forma di sfera. Anche se solitamente vengono utilizzate sfere perfette, il software permette di creare sfere irregolari con valori del raggio diversi lungo ognuno degli assi principali

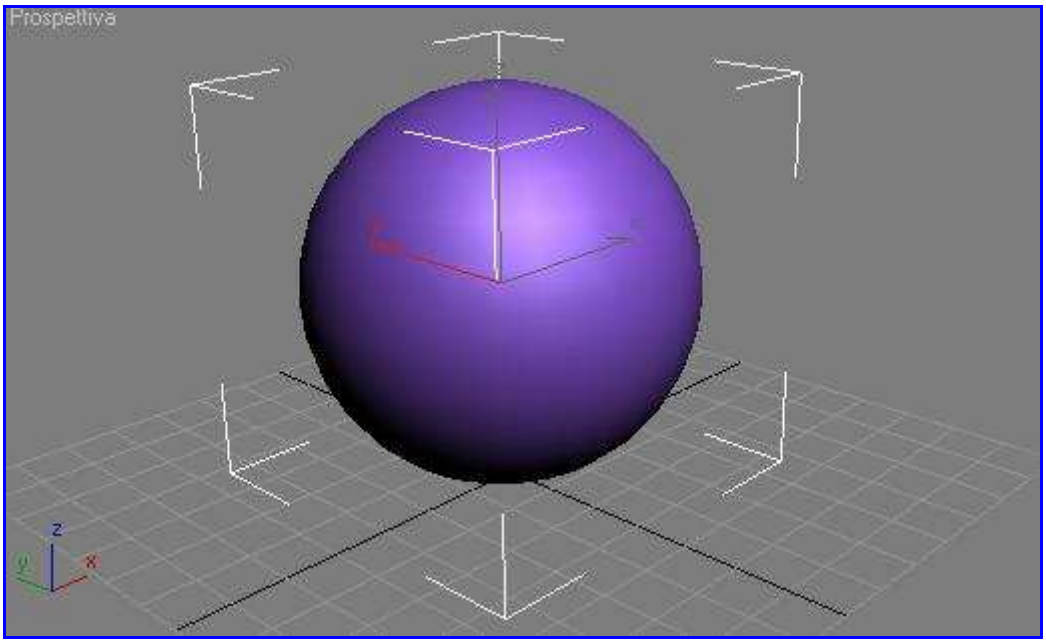

Figura 21 – La primitiva sphere

```
NewtonCollision* NewtonCreateCone(const NewtonWorld*
newtonWorld, dFloat radius, dFloat height, const dFloat
*offsetMatrix)
```

Permette di creare una primitiva a forma di cono. È possibile indicare le dimensioni del raggio alla base e l'altezza del cono.

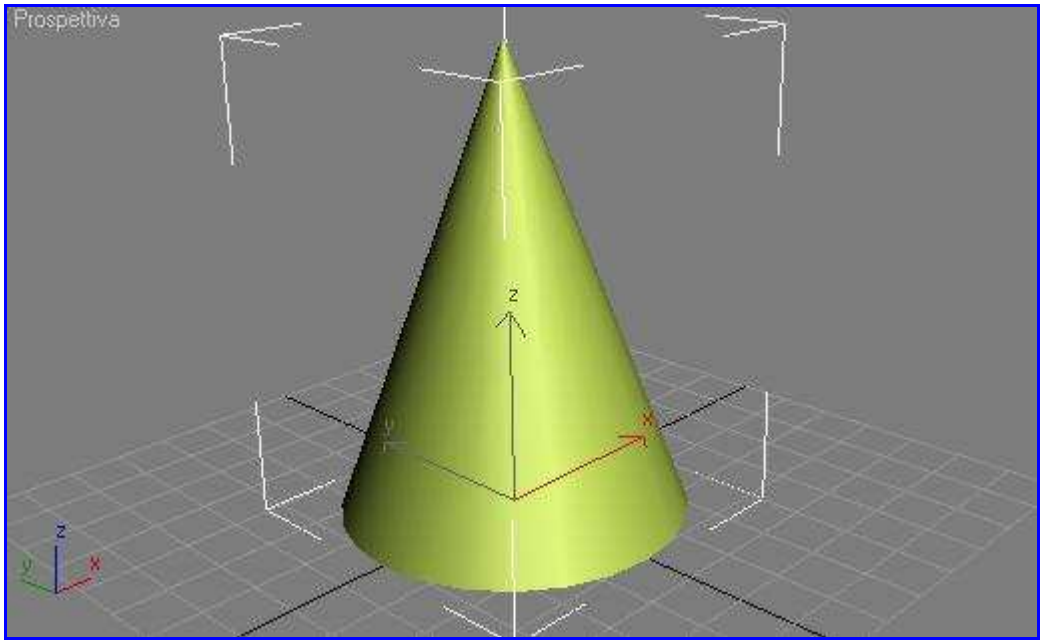

Figura 22 – La primitiva cone

```
NewtonCollision* NewtonCreateCapsule(const NewtonWorld*
newtonWorld, dFloat radius, dFloat height, const dFloat
*offsetMatrix)
```

Permette di creare una primitiva a forma di capsula. È posibile indicare l'altezza della capsula e il raggio. Questo tipo di primitiva è spesso utilizzato nei videogame per approssimare la geometria dei personaggi nella *collision detection*.

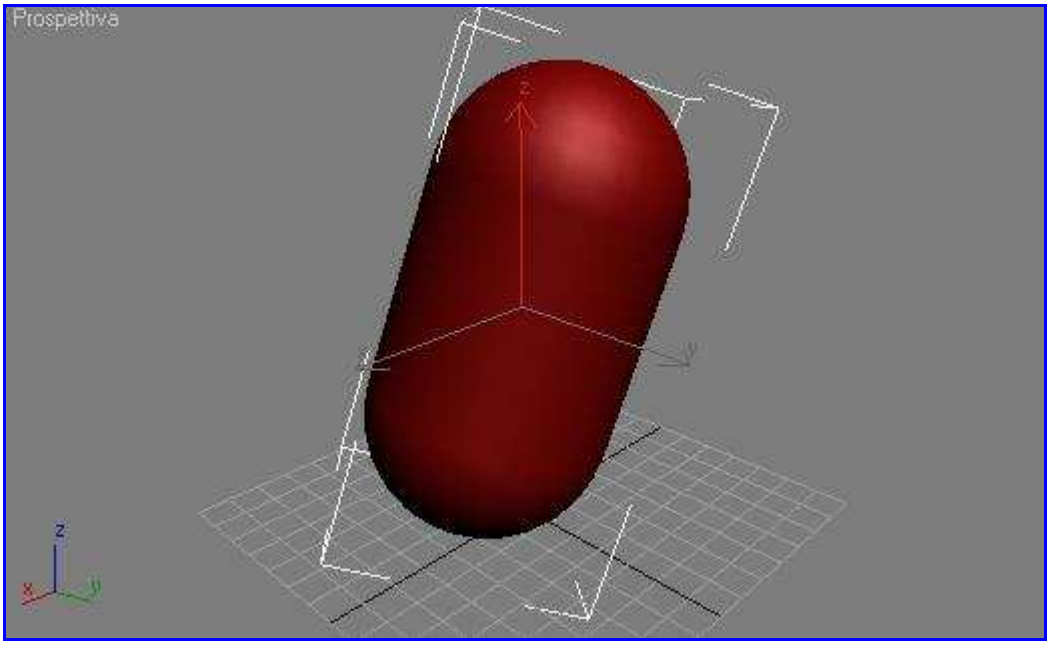

Figura 23 – La primitiva capsule

```
NewtonCollision* NewtonCreateCylinder(const NewtonWorld*
newtonWorld, dFloat radius, dFloat height, const dFloat
*offsetMatrix)
```

Permette di creare una primitiva a forma di cilindro. È possibile specificare il raggio di base e l'altezza del cilindro.

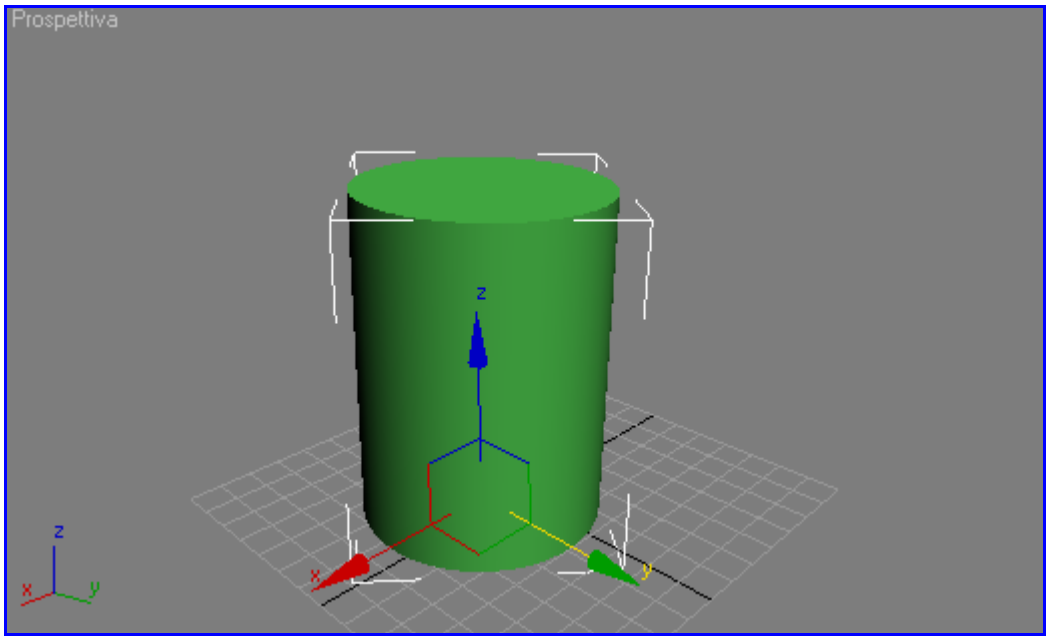

Figura 24 – La primitiva cylinder

```
NewtonCollision* NewtonCreateChamferCylinder(const
NewtonWorld* newtonWorld, dFloat radius, dFloat height, const
dFloat *offsetMatrix)
```

Permette di creare una primitiva a forma di cilindro smussato. È possibile indicare le dimensioni del raggio e l'altezza del cilindro.

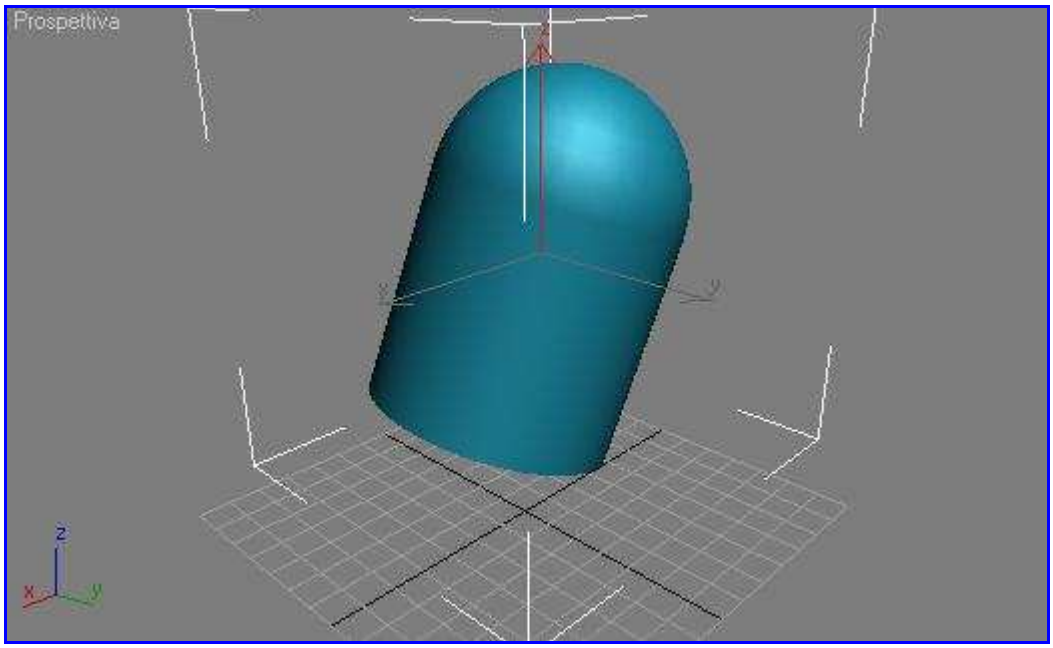

Figura 25 – La primitiva chamfer cylinder

```
NewtonCollision* NewtonCreateConvexHull(const NewtonWorld*
newtonWorld, int count, dFloat* vertexCloud, int
strideInBytes, dFloat *offsetMatrix)
```

Permette di creare un guscio convesso con una complessità arbitraria. I parametri da fornire sono: il numero di punti da cui è composto il guscio (minimo 4), un puntatore a un array che contiene i vertici del guscio, le dimensioni in byte di un singolo vertice (minimo 12 byte).

```
NewtonCollision* NewtonCreateCompoundCollision(const
NewtonWorld* newtonWorld, int count, NewtonCollision* const
collisionPrimitiveArray[])
```

Permette di creare una geometria composta da geometrie più semplici. I parametri da passare alla funzione sono il numero di geometrie semplici e un puntatore a un array che contiene i puntatori alle diverse geometrie.

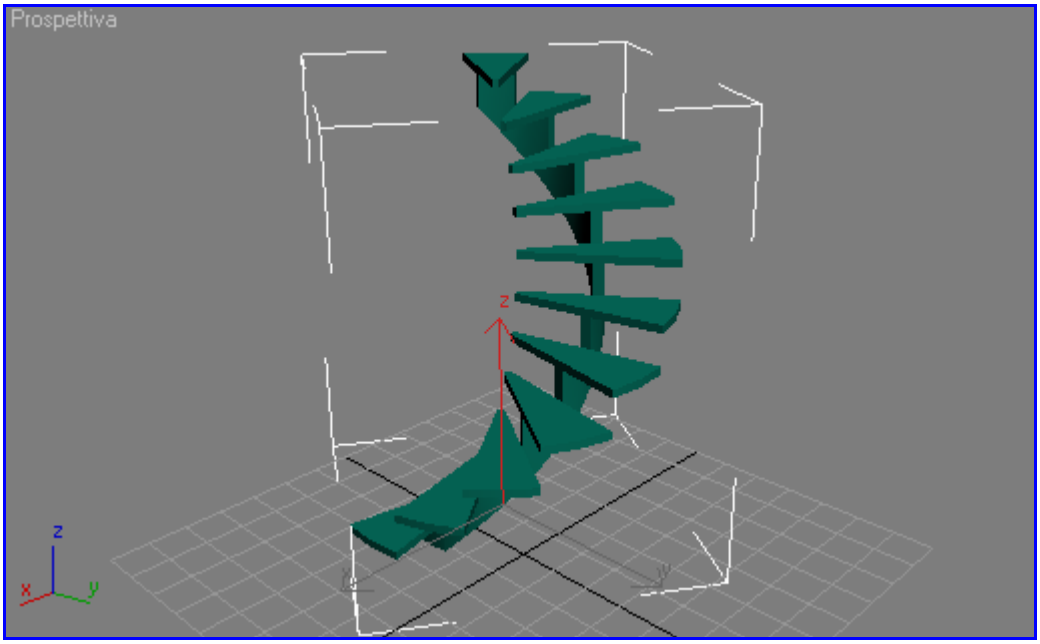

Figura 26 – Una scala a chiocciola composta da geometrie semplici

2.2.8 - Creazione dei corpi rigidi

Dopo aver creato le geometrie degli oggetti della simulazione è il momento di creare i corpi rigidi. Il corpo rigido possiede caratteristiche prettamente fisiche come la velocità, la massa, la distribuzione di massa e l'accelerazione. Tra le principali funzionalità troviamo le seguenti:

```
NewtonBody* NewtonCreateBody(const NewtonWorld* newtonWorld,
const NewtonCollision* collisionPtr)

void NewtonDestroyBody(const NewtonWorld* newtonWorld, const
NewtonBody* bodyPtr)
```

Con queste funzioni è possibile creare e distruggere un corpo rigido. Nella creazione di un corpo oltre al *Newton world* al quale dovrà appartenere si deve fornire un puntatore alla geometria che descrive il corpo stesso.

```
void NewtonBodySetUserData(const NewtonBody* bodyPtr, void*
userDataPtr)

void* NewtonBodyGetUserData(const NewtonBody* bodyPtr)
```

Spesso si desidera poter associare a un corpo rigido delle informazioni addizionali che possono servire per altri scopi della computazione. Si pensi ad esempio al caso classico di un video game in cui al corpo fisico che descrive il personaggio del gioco deve in qualche modo corrispondere una forma di visualizzazione a schermo: in questi casi è utile e conveniente legare in modo permanente i due oggetti. *Newton Game Dynamics* permette di effettuare questa operazione con le due funzioni di cui sopra che servono appunto per aggiungere un puntatore a una generica struttura dati definita dall'utente a un corpo rigido e

in seguito di recuperarla quando necessario. È ovviamente compito del programmatore eseguire il corretto casting per ottenere il tipo esatto di dato.

```
void NewtonBodySetTransformCallback(const NewtonBody*
bodyPtr, NewtonSetTransform callback)

void NewtonBodySetAutoactiveCallback(const NewtonBody*
bodyPtr, NewtonBodyActivationState callback)

void NewtonBodySetForceAndTorqueCallback(const NewtonBody*
bodyPtr, NewtonApplyForceAndTorque callback)

void NewtonBodySetDestructorCallback(const NewtonBody*
bodyPtr, NewtonBodyDestructor callback)
```

Queste (e molte altre) funzioni permettono di definire delle routine di *callback* da associare a ciascun corpo e che verranno richiamate al verificarsi di particolari eventi. Si va dalle routine per la trasformazione (traslazione, rotazione, scaling) di un oggetto alle funzioni che permettono di definire le forze da applicare a un corpo o di reagire alle collisioni. Tutto il motore fisico è sviluppato in modo da implementare numerose funzioni di *callback* che permettono una notevole varietà di comportamenti diversi il cui unico limite è la fantasia del programmatore.

```
void NewtonBodySetMassMatrix(const NewtonBody* bodyPtr,
dFloat mass, dFloat Ixx, dFloat Iyy, dFloat Izz)
```

Permette di impostare la matrice d’inerzia di un corpo rigido. I parametri richiesti sono il valore della massa e il momento d’inerzia rispetto agli assi principali di inerzia.

```
void NewtonBodySetMatrix(const NewtonBody* bodyPtr, const
dFloat* matrixPtr)
```

Questa funzione permette di settare la matrice di trasformazione di un corpo rigido.

```
void NewtonBodySetForce(const NewtonBody* bodyPtr, const
dFloat* forcePtr)

void NewtonBodyAddForce(const NewtonBody* bodyPtr, const
dFloat* forcePtr)

void NewtonBodySetTorque(const NewtonBody* bodyPtr, const
dFloat* forcePtr)

void NewtonBodyAddTorque(const NewtonBody* bodyPtr, const
dFloat* forcePtr)
```

Il gruppo di funzioni indicato sopra permette di settare e applicare forze e momenti torcenti a un oggetto. È importante far notare che queste funzioni possono essere richiamate solo all'interno della funzione di *callback NewtonBodySetForceAndTorqueCallback()* definita in precedenza dall'utente.

```
void NewtonBodySetCentreOfMass(const NewtonBody* bodyPtr,
const dFloat* comPtr)
```

Consente di indicare il centro di massa di un corpo rigido.

```
void NewtonBodySetCollision(const NewtonBody* bodyPtr, const
NewtonCollision* collisionPtr)
```

Permette di cambiare la geometria associata a un corpo.

```
void NewtonBodySetMaterialGroupID(const NewtonBody* bodyPtr,
int id)
```

Definisce il materiale per il corpo rigido. L'impiego dei materiali permette di semplificare la gestione delle collisioni tra oggetti; basta infatti definire in modo standard il comportamento dei materiali e il comportamento di caso di collisione tra ogni possibile coppia di materiali e poi assegnare il materiale desiderato all'oggetto per ottenere dei comportamenti uniformi.

```
void NewtonBodySetAutoFreeze(const NewtonBody* bodyPtr, int
state)

void NewtonBodySetFreezeTreshold(const NewtonBody* bodyPtr,
dFloat freezeSpeedMag2, dFloat freezeOmegaMag2, int
framesCount)
```

Queste funzioni permettono di implementare la possibilità di 'congelare' un corpo rigido che resti immobile per un determinato numero di frame consecutivi. Questa possibilità permette di risparmiare cicli di CPU aumentando così le prestazioni. Le due funzioni permettono rispettivamente di attivare la modalità automatica di congelamento e di settare i valori di soglia per considerare fermo un corpo.

```
void NewtonBodySetVelocity(const NewtonBody* bodyPtr, const
dFloat* velocity)

void NewtonBodySetOmega(const NewtonBody* bodyPtr, const
dFloat* omega)

void NewtonAddBodyImpulse(const NewtonBody* bodyPtr, const
dFloat* pointDeltaVeloc, const dFloat* pointPosit)
```

Queste funzioni permettono di impostare la velocità lineare e angolare di un corpo rigido, nonchè di applicare un impulso che cambi la quantità di moto dell'oggetto.

2.2.9 - Creazione dei vincoli tra i corpi

Se gli oggetti che vogliamo simulare sono composti da diverse parti che possono muoversi (si pensi ad esempio a una ruota composta dal cerchione e dal mozzo con il primo che è libero di ruotare attorno al secondo ma non di traslare in nessuna direzione) c'è bisogno di funzioni che ci permettano di creare vincoli tra i corpi rigidi. È da notare che per ogni vincolo è possibile definire una funzione di *callback* che permette di gestire ogni movimento dei corpi collegati. Vediamo le funzionalità più importanti fornite da *Newton Game Dynamics*:

```
NewtonJoint* NewtonConstraintCreateBall(const NewtonWorld*
newtonWorld, const dFloat* pivotPoint, const NewtonBody*
childBody, const NewtonBody* parentBody)

void NewtonBallSetConeLimits(const NewtonJoint* ball, const
dFloat* pin, dFloat maxConeAngle, dFloat maxTwistAngle)

void NewtonBallSetUserCallback(const NewtonJoint* ball,
NewtonBallCallBack callback)
```

Questo gruppo di funzioni permette di creare e gestire un vincolo di tipo *ball&socket*. Tra i parametri figurano il massimo angolo di cui possono ruotare e traslare i due corpi connessi rispetto al vincolo.

```
NewtonJoint* NewtonConstraintCreateHinge(const NewtonWorld*
newtonWorld, const dFloat* pivotPoint, const dFloat* pinDir,
const NewtonBody* childBody, const NewtonBody* parentBody)

void NewtonHingeSetUserCallback(const NewtonJoint* Hinge,
NewtonHingeCallBack callback)
```

Questo gruppo di funzioni permette di creare e gestire un vincolo di tipo *hinge*.

```
NewtonJoint* NewtonConstraintCreateSlider(const NewtonWorld*
newtonWorld, const dFloat* pivotPoint, const dFloat* pinDir,
const NewtonBody* childBody, const NewtonBody* parentBody)

void NewtonSliderSetUserCallback(const NewtonJoint* Slider,
NewtonSliderCallBack callback)
```

Questo gruppo di funzioni permette di creare e gestire un vincolo di tipo *slider*.

```
NewtonJoint* NewtonConstraintCreateUniversal(const
NewtonWorld* newtonWorld, const dFloat* pivotPoint, const
dFloat* pinDir0, const dFloat* pinDir1, const NewtonBody*
childBody, const NewtonBody* parentBody)

void NewtonUniversalSetUserCallback(const NewtonJoint*
Universal, NewtonUniversalCallBack callback)
```

Questo gruppo di funzioni permette di creare e gestire un vincolo di tipo *universale*.

```
NewtonJoint* NewtonConstraintCreateUpVector(const
NewtonWorld* newtonWorld, const NewtonCollision *pinDir,
const NewtonBody *body)
```

Questo gruppo di funzioni permette di creare un vincolo di tipo *upVector*. Questo tipo di vincolo permette a un corpo di traslare liberamente nello spazio, ma ne limita la rotazione a un solo asse specificato in fase di creazione del vincolo.

```
NewtonJoint* NewtonConstraintCreateUserJoint(const
NewtonWorld* newtonWorld, int maxDOF,
NewtonUserBilateralCallBack callback, const NewtonBody*
childBody, const NewtonBody* parentBody)

void NewtonUserJointAddLinearRow( const NewtonJoint*
joint,const dFloat *pivot0, const dFloat *pivot1, const
dFloat *dir)

void NewtonUserJointAddAngularRow( const NewtonJoint* joint,
dFloat relativeAngleError, const dFloat *pin)
```

Questo gruppo di funzioni permette di creare e gestire un vincolo definito dall'utente. Si parte da un generico vincolo con 6 gradi di libertà e poi si limitano le rotazioni e le traslazioni lungo gli assi e le direzioni desiderate. Come abbiamo visto *Newton Game Dynamics* fornisce diversi tipi di vincolo predefiniti, ma in realtà basterebbe questo unico tipo di vincolo per creare tutti gli altri.

```
NewtonRagDoll* NewtonCreateRagDoll(const NewtonWorld*
newtonWorld)

void NewtonRagDollBegin(const NewtonRagDoll* ragDoll)

void NewtonRagDollEnd(const NewtonRagDoll* ragDoll)

NewtonRagDollBone* NewtonRagDollAddBone(const NewtonRagDoll*
ragDoll, const NewtonRagDollBone* parent, void *userData,
dFloat mass, const dFloat* matrix, const NewtonCollision*
boneCollision, const dFloat* size)

void NewtonRagDollSetTransformCallback( const NewtonRagDoll*
ragDoll, NewtonSetRagDollTransform callback)

void NewtonRagDollSetForceAndTorqueCallback( const
NewtonRagDoll* ragDoll, NewtonApplyForceAndTorque callback)

void NewtonRagDollBoneSetLimits( const NewtonRagDollBone*
bone, const dFloat* coneDir,
dFloat minConeAngle, dFloat maxConeAngle, dFloat
maxTwistAngle, const dFloat* lateralConeDir, dFloat
negativeBilateralConeAngle, dFloat
positiveBilateralConeAngle)
```

Questo gruppo di funzioni semplifica la creazioni di oggetti di tipo *ragdoll*. Una *ragdoll* è una struttura composta da *bones* (ossa) legate tra loro da vincoli di tipo *ball&socket*. È comunemente utilizzata per simulare esseri umani o animali.

```
NewtonJoint* NewtonConstraintCreateVehicle(const NewtonWorld*
newtonWorld, const dFloat* upDir, const NewtonBody* body)

void NewtonVehicleSetTireCallback(const NewtonJoint* vehicle,
NewtonVehicleTireUpdate update)
```

```
void* NewtonVehicleAddTire(const NewtonJoint* vehicle, const
dFloat* localMatrix, const dFloat* pin, dFloat mass, dFloat
width, dFloat radius, dFloat suspesionShock, dFloat
suspesionSpring, dFloat suspesionLength, void* userData, int
collisionID)

void NewtonVehicleReset(const NewtonJoint* vehicle)

void NewtonVehicleRemoveTire(const NewtonJoint* vehicle,
void* tireId)

void NewtonVehicleSetTireSteerAngle( const NewtonJoint*
vehicle, void* tireId, dFloat angle)

void NewtonVehicleSetTireTorque( const NewtonJoint* vehicle,
void* tireId, dFloat torque)

void NewtonVehicleTireSetBrakeAcceleration(const NewtonJoint*
vehicle, void* tireId, dFloat acceleration, dFloat
maxFrictionTorque)
```

Questo gruppo di funzioni serve a gestire semplicemente un oggetto che replica le funzionalità di un veicolo. Le funzioni permettono di aggiungere ruote al veicolo e di sterzarle dell'angolo desiderato. Il motore fisico si occupa di gestire da solo aspetti come le sospensioni delle ruote, l'accelerazione e i freni del veicolo.

3 - Il modello BRR (Business Readiness Rating)

Decidere quali software adottare all'interno di un'organizzazione può essere un compito difficile. Insieme ai benefici promessi dai vari software vengono i rischi come problemi di compatibilità, usabilità, scalabilità e anche i problemi legali. Tradizionalmente le aziende dipendono dai software commerciali e proprietari a dispetto di limitazioni quali:

- **Costi.** Generalmente sono molto alti.
- **Codice sorgente non disponibile.** L'attuale livello di sicurezza e qualità del software è sconosciuto.
- **Lock-in e insufficienza di influenza sulla roadmap.** Il venditore sceglie quali miglioramenti apportare. Le richieste dei clienti possono essere ignorate se queste non sono contemplate dalla *roadmap* del venditore.

Questi fattori compensano il maggiore beneficio derivante dall'uso di software proprietario:

- **Supporto.** I produttori di software commerciale hanno staff di supporto dedicati ad aiutare i clienti a risolvere i loro problemi con il prodotto.

Oggi, l'uso di software Open Source in ambiente business è considerato in aumento. Le ragioni di questo incremento comprendono:

- **Costi.** Spesso si tratta di software gratuito.
- **Accesso al codice sorgente.** Considerato che il codice sorgente è aperto si hanno i benefici derivanti dalla revisione automatica del codice. Conseguentemente i progetti Open Source maturi tendono a essere più sicuri e ad avere un minor numero di bug rispetto alle loro controparti commerciali.

- **Architettura aperta.** Il software Open Source è spesso sviluppato da una comunità virtuale. Visto che lo sviluppo è spesso geograficamente molto distribuito i progetti Open Source sono generalmente progettati per essere modulari. Il codice modulare si può facilmente estendere ed è altrettanto semplice farne il debug..
- **Qualità.** I sorgenti aperti e l'architettura trasparente descritti sopra uniti a un progetto ben amministrato permettono di produrre software maturo e di alta qualità.

A dispetto di questi benefici certi problemi possono ostacolare l'adozione di software Open Source. Il *range* dei prodotti open va dai lavori individuali di bassa qualità a soluzioni di alta qualità realizzati da grandi aziende. Solo su *SorceForge* sono elencati più di 100000 progetti Open Source, senza contare tutti quelli presenti su altri repositories come *CodeHaus, Tigris, Java.net, ObjectWeb* e *OpenSymphony*.

I potenziali utenti si trovano ad affrontare le seguenti sfide:

- **Selezione.** Per certe categorie di software le scelte sono virtualmente illimitate.
- **Supporto.** Molti pacchetti Open Source non sono supportati in modo professionale.
- **Longevità.** Considerato che molti progetti non sono coperti da compagnie commerciali la disponibilità di futuri rilasci dipende dagli sforzi della comunità.
- **Volatilità.** Molti progetti Open Source seguono il paradigma del "rilasciare presto, rilasciare spesso" per ottenere credito nella comunità degli sviluppatori. Conseguentemente nel mondo dell'Open Source l'unica cosa costante è il cambiamento. Molti potenziali utenti non sono pronti a tracciare e implementare i rapidi aggiornamenti e cambiamenti del software.
- **Bassa qualità per i progetti immaturi.** Durante il suo concepimento, un progetto Open Source è di solito sviluppato da uno o pochi hobbysti che per passione decidono di creare qualcosa. Questi primi sviluppatori possono mancare delle risorse o dell'esperienza necessaria a sviluppare un prodotto completo. I progetti possono diventare orfani rapidamente dopo che gli sviluppatori iniziali sono riusciti a creare un componente che risolva i loro problemi. I nuovi progetti possono essere sviluppati senza alcun criterio formale di ingegneria del software o senza alcun test.

Come risultato c'è un ampio continuum nella qualità del software open. I progetti Open Source largamente adottati frequentemente evolvono in progetti software di alta qualità spesso in modo migliore rispetto alle loro controparti commerciali. Comunque i software aperti immaturi

possono offrire agli utenti più rischi che benefici. C'è un reale bisogno di un metodo standardizzato e largamente adottato per la stima della maturità dei prodotti Open.

3.1 - Le pratiche attuali di stima del software

Decidere quale software adottare generalmente implica l'elencare diverse alternative e poi fare una rapida stima per vagliarle e ridurle a una lista più contenuta che incontri i requisiti essenziali che richiediamo al software di cui abbiamo bisogno. Creare una lista ridotta può intimidire data la vastità di alternative Open Source esistenti e l'incertezza riguardo a future versioni. Eppure senza una lista ridotta troppo tempo ed energie andrebbero sprecate nella valutazione di progetti inadatti.

Una buona lista ridotta dovrebbe includere molte delle alternative fattibili ed escludere invece tutte quelle inappropriate. Per certe categorie di applicazioni certe proprietà del software o degli standard a cui questo aderisce possono rappresentare degli utili filtri, come la lingua principale dell'applicazione o i tipi di database supportati. Comunque per molte categorie di applicazioni trovare i giusti filtri può essere difficile. Ad esempio usando un semplice filtro come "linguaggio di programmazione" i risultati che troviamo possono essere comunque troppo numerosi.

Dopo aver creato la shortlist, lo staff di programmatori deve valutare più a fondo ciascuno dei software rimasti sulla lista. Spesso più valutatori devono creare i propri metodi di stima, senza avere accesso ai dati o ai metodi di stima precedenti dei propri pari, e poi deve ri-stimare da solo i software della lista.

In pratica, molte valutazioni di progetti software sono fatte *ad hoc* senza un processo di stima formale. I metodi ad hoc possono essere scorretti o incompleti nelle loro stime ed è estremamente difficile validare la correttezza della valutazione. Un meccanismo di valutazione inaccurato e incompleto può indurre in decisioni e scelte di prodotti scorrette, il che fa dei metodi di stima ad hoc dei metodi molto rischiosi.

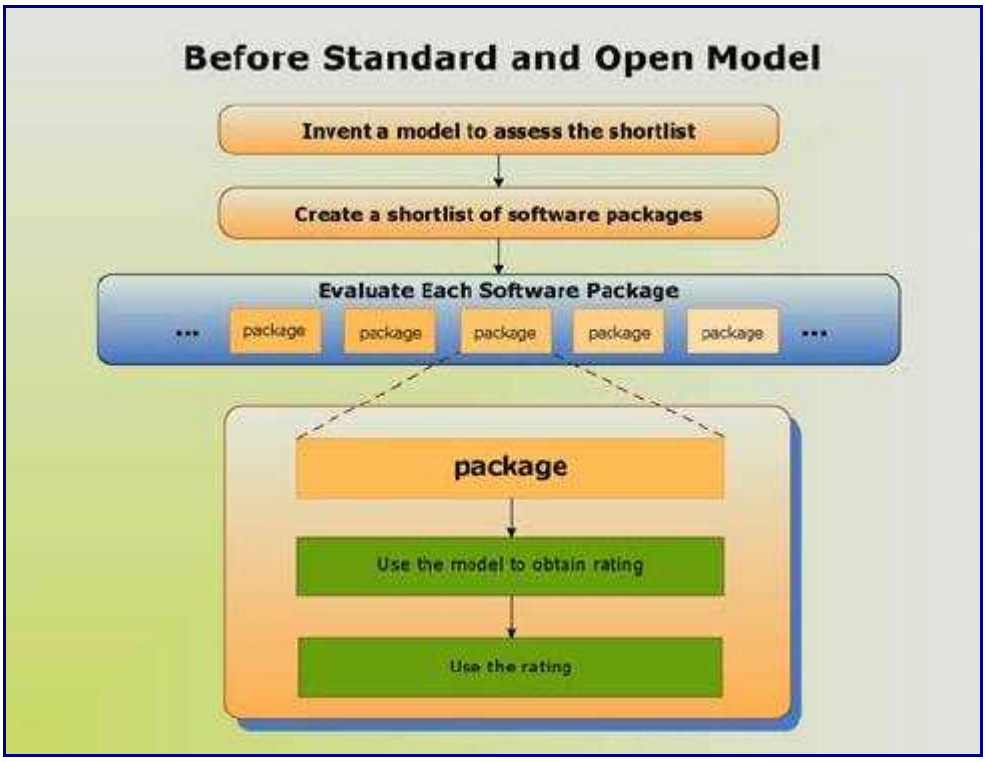

Figura 27 – Il generico processo di valutazione di un software

3.2 - Un modello aperto e standard per la valutazione del software

Come possono gli utenti business, gli sviluppatori o gli ingegneri del software decidere più facilmente quale software Open Source utilizzare? Come possono fiduciosamente determinare se il software che stanno considerando è abbastanza maturo e *business ready* per i propri scopi?

Vediamo di proporre un modello di stima **aperto e standard** che incrementi la facilità e la correttezza della valutazione e acceleri l'adozione di software Open Source.

In aggiunta un modello di stima aperto e standard che sia largamente adottato e non controverso dovrebbe permettere agli utenti di software Open Source di condividere i risultati delle stime.

Perché standard? Un modello standardizzato consente accordi comuni sulle stime dei punteggi. Perché aperto? Un modello aperto promuove la fiducia nel processo di stima. Questo inoltre assicura che il modello di stima è flessibile, sempre nel rispetto di futuri cambiamenti. La

validazione della correttezza di un modello di stima è chiara e semplice: i potenziali utenti del modello possono guardarlo, commentarlo e migliorarlo.

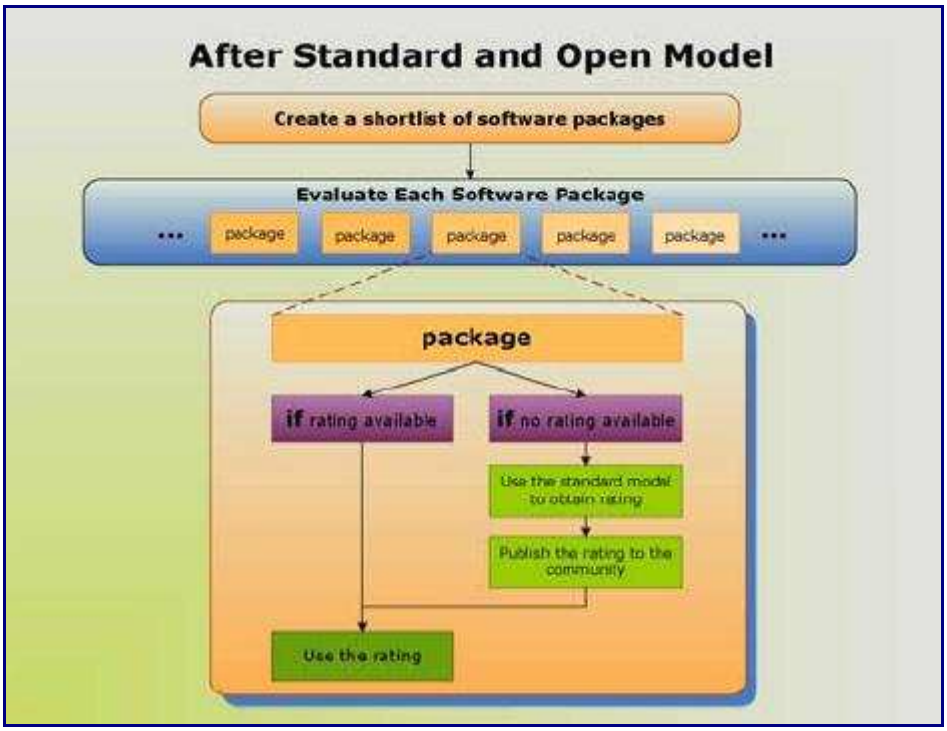

Figura 28 – Modello di valutazione standard e aperto

Un tale modello dovrebbe includere tutti i requisiti cruciali di un buon modello di classificazione del software: dovrebbe essere completo, semplice, adattabile e consistente (CSAC):

- **Completo.** Il requisito primario di ogni modello di valutazione di un prodotto è la capacità del modello di evidenziare ogni caratteristica preminente del prodotto, come il fatto che sia vantaggioso o no. Questo è necessario perché la valutazione di un qualunque prodotto non sia fuorviante.
- **Semplice.** Per acquistare un largo consenso il modello deve essere facile da capire e relativamente semplice da usare. Inoltre la classificazione e la terminologia dovrebbe essere amichevole per l'utente. Comunque la completezza del modello ha una priorità maggiore.
- **Adattabile.** Dati i rapidi cambiamenti nell'industria del software ogni modello di valutazione creato oggi potrebbe essere inutilizzabile in futuro. Durante la fase di concezione è impossibile catturare ogni potenziale uso del modello; conseguentemente

bisogna sforzarsi per costruire un modello con in testa l'obiettivo dell'adattabilità e cercando di tenerlo aperto. In questo modo quando il modello richiede un'estensione sarà facile aggiungerla senza stravolgere il progetto attuale.

- **Consistente.** La scala e il punteggio prodotti dal modello dovrebbero essere consistenti anche per differenti usi del modello. Una valutazione uguale per due software diversi appartenenti a diverse categorie dovrebbe comunque significare un uguale maturità dei software.

3.3 - Precedenti modelli per la stima dei software Open Source

Esistono diversi approcci per la valutazione del software. Almeno due precedenti iniziative miravano a offrire agli utenti una metodologia per la stima e la valutazione sull'idoneità dei software Open Source: L' *Open Source Maturity Model* sviluppato da Bernard Golden di Navicasoft e pubblicizzato nel suo libro *Succeeding with Open Source* e il *CapGemini Open Source Matutity Model*, disponibile su seriouslyopen.org. Questi modelli sono entrambi degli eccellenti lavori pionieristici nel settore della valutazione dei software.

Volendo proporre un nuovo modello si dovranno usare concetti simili per una completa valutazione del software, ancora dovremo fornire una valutazione più dettagliata dei dati e dei punteggi per stimare la maturità del software, specialmente per gli aspetti operativi e di supporto. Distinguere particolari aree di stima e fornire una valutazione pesata basata sull'uso specifico sono alcuni dei concetti comuni presentati dai due modelli precedenti. Qualcuno ha provato a estendere questi concetti mirando a fornire un modello scientifico per la classificazione che contiene un percorso chiaro che va dalla valutazione dei dati all'assegnamento dei voti fino alla valutazione finale. Chi ha creato questo modello desiderava espandere l'idea di una metodologia per la stima del software in una maggiormente adottabile e facile da usare nel maggior numero possibile di situazioni in cui servisse un metodo di stima.

3.4 - Introduzione al Business Readiness Rating Model (BRR)

Il modello *Business Readiness Rating* (BRR) intende aiutare i manager del settore informatico a stimare quali software Open Source potrebbero essere maggiormente indicati per i loro bisogni. Anche gli utenti dei software aperti possono condividere le proprie valutazioni come potenziali *adopters* continuando così *l'architettura di partecipazione* tipica dell'Open Source.

In questo modello vengono offerte proposte per standardizzare differenti tipi di dati di valutazione e per raggrupparli in categorie. Per consentire l'adozione di questo modello di stima per ogni requisito d'uso che il software può incontrare il processo viene separato in quattro fasi.

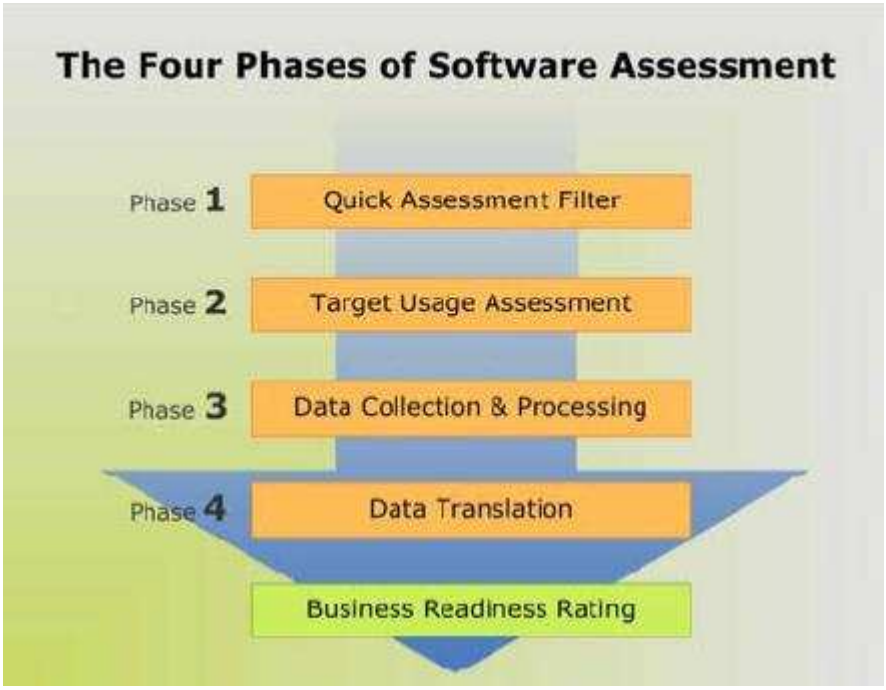

Figura 29 – Le 4 fasi della stima del modello BRR

Per prima viene una fase di stima rapida per includere o escludere pacchetti software per creare una shortlist di candidati fattibili. La seconda fase consiste nel classificare l'importanza delle categorie di stima e delle metriche usate per valutare ciascuna categoria. La terza fase consiste nel raccogliere i dati veri e propri assegnando dei punteggi per ogni metrica individuata; la quarta e ultima fase consiste nell'eleborazione dei dati prodotti e nella traduzione in un giudizio di *business*

readiness. Nel modello BRR un componente software è valutato con un punteggio che va da 1 a 5 dove 1 sta per *inaccettabile* e 5 sta per *eccellente*.

3.4.1 - Filtro rapido

Per stimare la maturità di un software Open Source l'utente può cominciare guardando a diverse proprietà quantitative e qualitative dei componenti in esame. Durante la fase iniziale di stima rapida un semplice filtro permette agli utenti di includere o scartare con fiducia i software in una shortlist. Alcuni esempi di filtri utilizzabili in questa fase possono essere:

- Che tipo di licenza d'uso ha il software
- Aderisce agli standard?
- Ci sono aziende di riferimento che già utilizzano il software?
- C'è un'organizzazione stabile dietro lo sviluppo del software?
- In che linguaggio è realizzato?
- Qual è la lingua del software? È possibile trovare la propria lingua?
- Esistono libri pubblicati su questo software?

Questa lista di criteri di filtro per la stima rapida non è certamente esaustiva. Gli utenti possono e anzi dovrebbero aggiungere i filtri che si ritengono più importanti per il particolare software in esame.

3.4.2 - Metriche e categorie

Dopo aver completato la prima fase è importante vedere quali metriche e categorie usare per una fase di stima che vada più in profondità.

Definiamo una proprietà misurabile di un software Open Source con il nome di *metrica.* Alcuni esempi possono essere: il numero di libri pubblicati sul software, il numero di committenti del progetto, il livello delle attività di test. Per creare un metodo standardizzato è importante normalizzare i dati grezzi di queste metriche. Le metriche quantitative come il numero di

downloads di un pacchetto software sono relativamente semplici da normalizzare. Anche le metriche qualitative hanno bisogno di essere normalizzate. È importante organizzare il processo di stima in aree di interesse, o *categorie di stima.* BRR definisce 12 categorie per la stima del software:

Categoria di stima	Descrizione
Funzionalità	Quanto bene il software incontra i requisiti d'uso dell'utente medio?
Usabilità	Quanto è buona l'interfaccia utente? Quanto è facile l'uso per l'utente finale?Quanto sono facili l'installazione, la configurazione, il deployment e la manutenzione?
Qualità	Qual è la qualità del design, del codice e dei test? Quanti errori vi sono nel codice?
Sicurezza	Qual è il grado di sicurezza del software?
Prestazioni	Quanto bene funziona il software?
Scalabilità	Quanto bene reagisce il software a un largo impiego?
Architettura	Quanto è modulare, portabile, estensibile, aperto e facilmente integrabile il software?
Supporto	Quanto è supportato il software?
Documentazione	Quanta documentazione esiste sul software? Di quale qualità?
Adozione	Quanto è adottato il software dalla comunitò e dal mercato?
Community	anto è ampia e attiva la community?
Professionalità	Qual è il grado di professionalità di chi sviluppa il progetto?

Si definisce la stima di uno specifico aspetto di un software come una *valutazione di categoria.* Una valutazione di categoria è ottenuta raggruppando insieme tutte le metriche che misurano lo stesso aspetto. Il metodo con cui il punteggio di una categoria viene calcolato può essere diverso per ogni categoria, ma il risultato deve usare la stessa scala (da 1 a 5). Una stessa

metrica può contribuire a più categorie in differenti modi: ad esempio Fedora rilascia una nuova versione ogni 6 mesi: questo può indicare un alto livello della vitalità della community ma un basso livello di stabilità.

3.4.3 - Orientamento operativo su misura

Nel modello BRR le classificazioni delle categorie possono avere diversi livelli di importanza a seconda dei requisiti d'uso richiesti dall'utente. I requisiti d'uso possono essere derivati da due fattori:

- **Dal tipo di componente**. (mail transfer agent, web container, browser web, suite pe l'ufficio, ecc.) .
- **Dalle configurazioni d'uso**. Ecco alcune configurazioni d'uso per le quali I software possono essere valutati:

  - ✓ **Uso mission-critical.** Il software deve lavorare per 24 ore al giorno 7 giorni su 7. Il business della compagnia dipende del tutto o in gran parte dai servizi offerti dal software ai clienti. Esempio: un server web.
  - ✓ **Uso di routine.** Il software è usato internamente all'azienda. Piccole attese per gli aggiornamenti sono accettabili senza avere un grande impatto sulla produttività dell'azienda.
  - ✓ **Sviluppo interno.** Un gruppo di sviluppo può valutare un software Open Source da integrare con il sistema interno per usi interni o per l'inclusione in un servizio o un prodotto per i clienti. Il gruppo di sviluppo può modificare il codice sorgente.
  - ✓ **Sperimentazione.** Inchieste sul software senza l'obiettivo primario di utilizzarlo per la produttività interna. Esempi includono studi comparativi di prodotti simili, l'esplorazione di moduli dei componenti software per studiarne i dettagli implementativi.

La giustapposizione del tipo di componente con le sue configurazioni d'uso ha un impatto significativo sul suo grado di maturità. Definiamo la combinazione di questi fattori come *orientamento operativo del software.* Sebbene la classificazione di categorie come documentazione, qualità del codice o livello di adozione possano fornire agli utenti numeri normalizzati e standardizzati facili da capire, questi numeri possono significare cose differenti per differenti

orientamenti operativi e requisiti d'uso. Un componente software Open Source può avere differenti gradi di maturità per differenti scopi d'uso. Un pacchetto software può essere considerato pronto per l'uso ordinario all'interno della azienda ma non ancora per un uso *mission-critical*.

Il grado di maturità di un componente software è calcolato pesando appropriatamente una collezione di valutazioni di categoria in accordo col tipo di software e il suo uso. Ciascun area di orientamento operativo si focalizza solo su un limitato numero di categorie di valutazione. La valutazione del grado di maturità per l'orientamento operativo del software è calcolata basandosi solo sulle categorie di valutazione considerate. Questo ci assicura che la valutazione del grado di maturità riflette le categorie più essenziali per la stima.

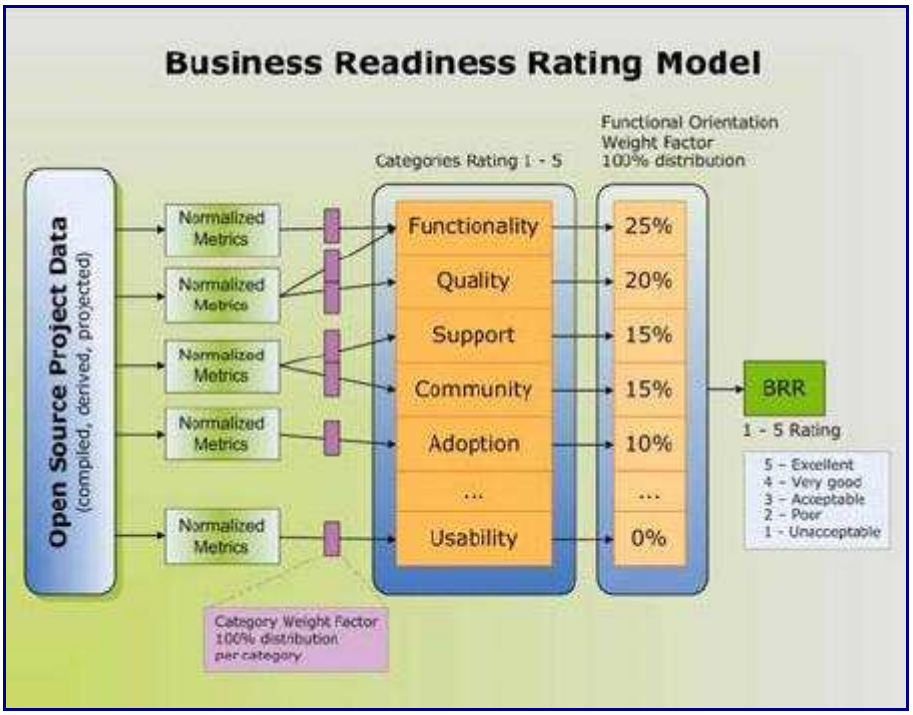

Figura 30 – I diversi passaggi del modello BRR

3.4.4 - Usare il modello

La fase di stima rapida e la definizione e classificazione di metriche e categorie in accordo con la loro importanza per l'orientamento operativo del software ci conduce alla fase in cui traduciamo i dati prodotti per calcolare il grado di maturità del software.

Fase 1 – Stima rapida

- Identificare una lista di software da valutare
- Misurare ciascun componente con dei criteri rapidi di stima
- Rimuovere ogni componente che non soddisfa i requisiti d'uso dalla lista

Fase 2 – Stima degli obiettivi d'uso

Pesi delle categorie

- Classificare le 12 categorie in accordo con la loro importanza (1 – alta, 12 – bassa)
- Prendere le prime 7 (o meno) categorie e assegnare a ciascuna una percentuale d'importanza, in modo tale che il totale dia 100%

Pesi delle metriche

- Per ciascuna metrica dentro ogni categoria classificarle in accordo con l'importanza per il grado di maturità.
- Per ciascuna metrica dentro ogni categoria assegnare una percentuale d'importanza in modo tale che il totale per ogni categoria dia 100%

Fase 3 – Collezione dei dati e calcolo

- Raccogliere i dati per ciascuna metrica usata in ogni categoria e calcolare i pesi applicati per ogni metrica

Fase 4 – Traduzione dei dati

- Usare le valutazioni delle categorie e l'orientamento operativo pesati per calcolare il punteggio di *Business Readiness Rating*.

4 - Applicazione del modello BRR

4.1 - Filtro veloce

Iniziamo ad applicare il modello BRR prendendo in esame una serie di software dedicati alla simulazione fisica trovati su internet ed evidenziando alcune caratteristiche peculiari di ciascuno di essi. Al termine faremo delle breve considerazioni e ridurremo il numero dei candidati a due.

Nella tabella sottostante sono riportati i risultati della nostra ricerca preliminare.

Nome	**Sito**	**Dimensione**	**Sistema operativo**	**Licenza**	**Linguaggio**
Open Dynamics Engine	www.ode.org	3.5 Mb	Windows	free	C/C++
Newton Game Dynamics	www.newtondynamics.com	14/15 Mb	Windows MacOS Linux	free	C/C++
Tokamak	www.tokamakphysics.com	2.0 Mb	Windows	free	C++
PhysX	www.ageia.com	35.0 Mb	Windows	free	C++
Havoc	www.Havoc.com		Windows	Libero per 6 mesi	C++
nV Physics	www.thephysicsengine.com	1.0 Mb	Windows	free	C++
True Axis	www.trueaxis.com	1.4 Mb	Windows	Free per uso non commerciale	C++

I software presi in considerazione

Decidiamo di prendere in considerazione i due software *Open Dynamics Engine* e *Newton Game Dynamics*. *Havoc* e *True Axis* infatti presentano delle limitazioni sulla licenza d'uso, *nV Physics* e *Tokamak* non sembrano avere un grande supporto da parte della comunità dei programmatori mentre *PhysX* sfrutta appieno le sue potenzialità solo in combinazione con la scheda hardware di *Ageia*.

4.2 - Assegnazione dei pesi

Ora dobbiamo decidere i criteri in base ai quali effettueremo le nostre valutazioni. Il modello BRR suggerisce dodici diverse categorie alle quali assegnare un peso percentuale in modo tale che la somma di tutti i pesi dia il 100%. Poi, per ogni categoria dovremo decidere delle metriche di valutazione che ci permettano di valutare in modo possibilmente preciso la categoria in oggetto. Per ognuna di queste metriche dovremo decidere un ulteriore peso secondo le stesse modalità usate per le categorie in modo da bilanciare la valutazione secondo i nostri effettivi bisogni. Le categorie essenziali proposte dal modello sono: funzionalità, usabilità, qualità, prestazioni, supporto, architettura, scalabilità, documentazione, adozione, community, professionalità e sicurezza. Le metriche per ciascuna categoria invece non sono standard ma possono variare in funzione del tipo di software da valutare.

4.2.1 - Assegnazione dei pesi per le categorie

Questa è la ripartizione dei pesi per le diverse categorie che ho deciso di adottare:

- Funzionalità 25%
- Prestazioni 20%
- Scalabilità 15%
- Usabilità 10%
- Qualità 10%
- Architettura 5%
- Supporto 5%
- Documentazione 5%
- Adozione 5%
- Community 0%
- Professionalità 0%
- Sicurezza 0%

È da notare che non è necessario assegnare un peso a ciascuna categoria specialmente se alcune di queste non sono fondamentali per l’uso che vogliamo fare del software.

4.2.2 - Assegnazione dei pesi per le diverse metriche

Ecco le metriche che ho deciso di utilizzare per ogni categoria e il peso assegnato:

Funzionalità

- Verosimiglianza della simulazione 20%
- Rilevamento collisioni 10%
- Creazione di geometrie semplici 10%
- Gestione collisioni 10%
- Distribuzione di massa 10%
- Resa dell'attrito 10%
- Applicazione di forze 5%
- Integratori 5%
- Creazione di geometrie complesse 5%
- Vincoli 5%
- Funzioni di supporto 5%
- Ottimizzazioni 5%

Prestazioni

- Numero di FPS (*Frame Per Second*) 40%
- Utilizzo CPU 30%
- Utilizzo memoria 30%

Scalabilità

- Valutazione generale sulla scalabilità del sistema basata su diverse prove pratiche 100%

Usabilità

- Facilità di programmazione 20%
- Facilità di reperimento 15%
- Facilità di integrazione 15%
- Facilità di deployment 15%
- Facilità di manutenzione 15%
- Download 10%
- Installazione 10%

Qualità

- Rilasci negli ultimi 12 mesi 50%
- Patch uscite negli ultimi 12 mesi 50%

Architettura

- Possibilità di modifiche a *runtime* 40%
- Possibilità di usare *API* 40%
- Utilizzo di *plugin* di terze parti 20%

Supporto

- Presenza di un help professionale 40%
- Volume di messaggi della mailing list 30%
- Altre forme di supporto 30%

Documentazione

- Materiale disponibile 100%

Adozione

- Numero di progetti 70%
- Numero di libri dedicati 30%

Community

- Nessuna metrica presa in considerazione

Professionalità

- Nessuna metrica presa in considerazione

Sicurezza

- Nessuna metrica presa in considerazione

4.3 - Collezione dei dati

4.3.1 - Scenario di valutazione

Per valutare alcuni aspetti del software sono stati adottati alcuni scenari operativi. In particolare sono state realizzate due diverse simulazioni 3D: la prima riguarda la simulazione di un asteroide che colpisce un palazzo, la seconda tenta di riprodurre il famoso gioco del domino. Per entrambe le simulazioni sono state realizzate due versioni, una che utilizza come motore fisico *Open Dynamics Engine* e una che invece utilizza *Newton Game Dynamics.* Tutte e quattro le applicazioni sono realizzate in C++ che è il linguaggio supportato da entrambi i software in esame e per il rendering video si basano sull'utilizzo di un altro prodotto Open Source: il motore grafico 3D *Irrlicht Engine*.

La simulazione del palazzo mette alla prova principalmente le capacità dei due software di riprodurre fedelmente le collisioni tra corpi, di simulare le forze di attrito tra pilastri e solette e di gestire la riproduzione di vincoli tra i corpi rigidi (figura 31).

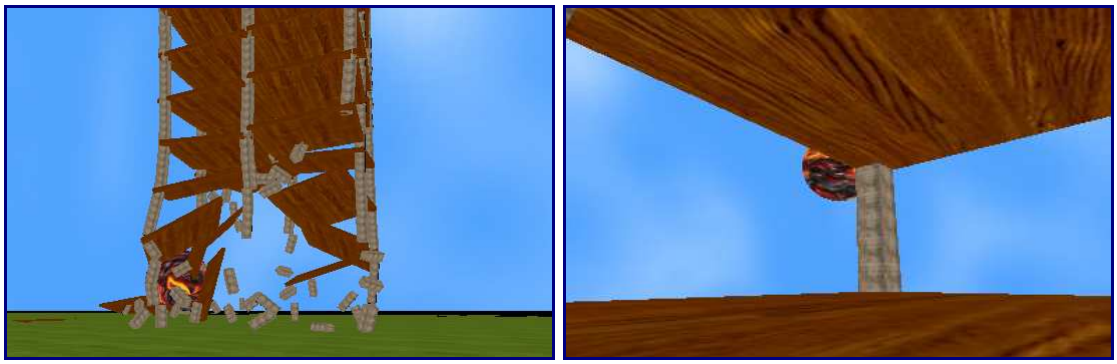

Figura 31 - Immagini della simulazione del palazzo

La simulazione del domino serve invece a testare aspetti che riguardano maggiormente aspetti dell'architettura del sistema: in particolare serve a valutare la scalabilità del sistema su un numero variabile di tessere e la possibilità di applicare ottimizzazioni come il "congelamento" dei corpi rigidi che restano inattivi per un certo periodo di tempo (figura 32).

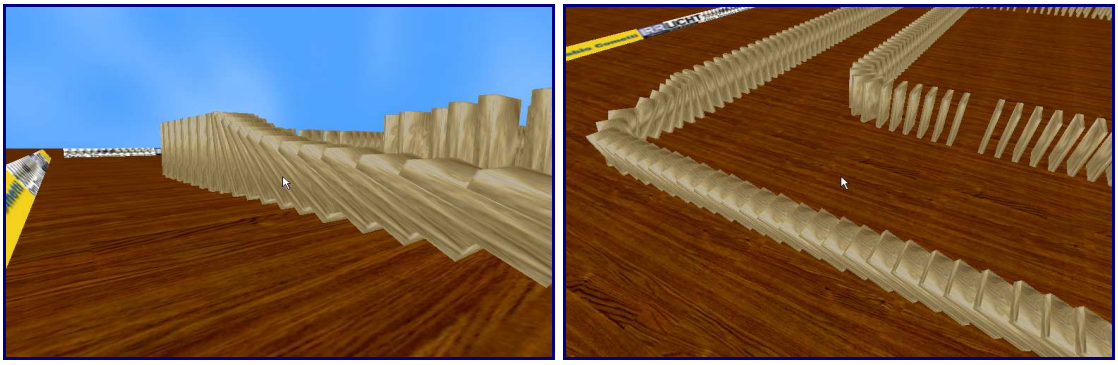

Figura 32 – Immagini della simulazione del domino

4.3.2 - Funzionalità

Nella categoria delle funzionalità rientrano tutti quei parametri che hanno direttamente a che fare con ciò che noi richiediamo direttamente al software.

Aspetto fisico	Open Dynamics Engine	Newton Game Dynamics
Verosimiglianza della simulazione	***	****
Applicazione di forze	****	**
Integratori	***	****
Rilevamento collisioni	****	****
Gestione collisioni	**	****
Creazione di geometrie semplici	***	***
Creazione di geometrie complesse	***	***
Distribuzione di massa	***	****
Resa dell'attrito	**	****
Vincoli	***	***
Funzioni di supporto	****	***
Ottimizzazioni	****	***

- **Verosimiglianza della simulazione:** Indica la resa globale della simulazione, quanto appare realistica rispetto all'esperienza reale. *Newton Game Dynamics* si dimostra lievemente superiore rispetto ad *Open Dynamics Engine.*
- **Applicazione di forze:** Indica la possibilità di applicare più o meno facilmente forze e momenti torcenti ad un corpo rigido. *Newton Game Dynamics* è limitato all'applicazione di un vettore forza le cui componenti indicano la forza applicata rispetto ai tre assi *X, Y, Z* rispetto al centro di massa del corpo e all'applicazione di momenti torcenti sempre lungo i tre assi principali. *ODE* permette di applicare forze e momenti in punti diversi dal centro di massa ed inoltre calcola automaticamente (se richiesto) la forza da applicare per simulare la gravità.
- **Integratori:** Indica la disponibilità di diversi integratori per la soluzione delle equazioni del moto dei corpi rigidi. *Newton Game Dynamics* permette di usare tre diversi integratori secondo le necessità: uno esatto, uno adattivo ed uno lineare. *ODE* permette di usare due integratori: uno più preciso ed uno che permette di risparmiare sia tempo di calcolo sia memoria; tuttavia entrambi non sembrano raggiungere il grado di precisione di *Newton Game Dynamics*.

- **Rilevamento collisioni:** Rappresenta la qualità con cui sono rilevate le collisioni tra e le interazioni tra diverse geometrie. Entrambi i software rispondono molto bene da questo punto di vista.
- **Gestione collisioni:** Indica con quale semplicità si possono gestire le collisioni tra i diversi oggetti del sistema. *ODE* si limita a fornire una funzione di *callback* che il programmatore deve sviluppare e configurare secondo le diverse coppie di oggetti in collisione. *Newton Game Dynamics* invece permette di definire dei materiali e il comportamento predefinito per la collisione tra due materiali diversi; è poi possibile assegnare un materiale ad ogni corpo rigido per la gestione degli urti, inoltre per ogni corpo è possibile definire una funzione di *callback* personalizzata.
- **Creazione di geometrie semplici:** Indica la semplicità con cui si possono definire geometrie semplici come parallelepipedi o sfere. Entrambi i software sostanzialmente mettono a disposizione le stesse primitive geometriche.
- **Creazione di geometrie complesse:** Indica la facilità con cui è possibile rappresentare geometrie complesse. Entrambi i motori fisici permettono di creare geometrie composte di *mesh* triangolari, ma il supporto di *Newton* appare più completo e più semplice da utilizzare. *ODE* permette però di assemblare più geometrie semplici in un unico corpo libero in modo più semplice rispetto a *Newton*. In sostanza i due software di equivalgono anche da questo punto di vista.
- **Distribuzione di massa:** indica la capacità di creare distribuzioni di massa che rappresentano i momenti d'inerzia delle geometrie presenti nel sistema. Entrambi i software permettono di definire una matrice d'inerzia arbitraria. Inoltre ODE permette di creare la distribuzione di massa automaticamente per le geometrie semplici, mentre per quelle complesse prevede un complicato sistema di traslazione delle masse. Newton invece mette a disposizione delle funzioni che calcolano automaticamente la matrice d'inerzia dalla geometria e dalla massa del corpo: sebbene tale funzionalità è meno intuitiva per le geometrie semplici si dimostra notevolmente più utile per le geometrie complesse.
- **Resa dell'attrito:** Rappresenta il grado di verosimiglianza con cui è riprodotto l'attrito tra due corpi. *ODE* permette di definire un attrito lungo la direzione principale del moto e uno lungo la direzione ortogonale. Entrambe le funzionalità però non sono riprodotte in maniera molto fedele. Al contrario *Newton* permette di definire coefficienti di attrito statico e dinamico per le diverse coppie di materiali e inoltre permette di simulare l'attrito sia in modo esatto sia approssimato; entrambe le modalità forniscono una resa più che sufficiente.

- **Vincoli:** indica la disponibilità di diverse tipologie di vincoli per riprodurre le relazioni tra i corpi rigidi. Entrambi i software permettono di utilizzare sostanzialmente le stesse funzionalità.
- **Funzioni di supporto:** Indica la disponibilità di funzioni che aiutano nella creazione del mondo fisico in cui si svolge la simulazione. Entrambi i motori offrono buone soluzioni, anche se *ODE* ne possiede un numero lievemente superiore.
- **Ottimizzazioni:** Rappresenta la possibilità di utilizzare funzioni che aiutano a migliorare le prestazioni sia in termini di velocità sia di precisione. Entrambi i software permettono di disabilitare i corpi rigidi che non sono momentaneamente interessati dalla simulazione; *ODE* inoltre permette di configurare due indicatori particolari chiamati ERP (*error reduction parameter*) e CFM (*constraints force mixing*) che migliorano la precisione della simulazione senza la necessità di cambiare il tipo d'integratore utilizzato.

4.3.3 - Usabilità

L'usabilità del software risponde a domande del tipo: *quanto è valida l'interfaccia utente?* Quanto è facile per l'utente finale usare il software? Quanto è facile installarlo? È facilmente integrabile? È facilmente mantenibile? Sono richieste operazioni speciali per il deployment?

Aspetto d'usabilità	Open Dynamics Engine	Newton Game Dynamics
Facilità di reperimento	****	*****
Download	*****	***
Installazione	****	*****
Facilità di programmazione	***	****
Facilità d'integrazione	****	****
Facilità di manutenzione	****	****
Deployment	*****	*****

- **Facilità di reperimento:** Indica quanto è facile trovare il software sulla rete e quanto è facile arrivare al download dello stesso. *Newton Game Dynamics* permette di arrivare al download direttamente dal proprio sito web, mentre *ODE* al contrario presenta dei collegamenti a *SourceForge.net* sul quale si trova direttamente l'intero progetto.
- **Download:** Indica la facilità di eseguire il download del software. Per entrambi i software non servono registrazioni per avere accesso al software; gioca a favore di *ODE* la dimensione del file di soli 3,5 Mb al confronto dei 15 Mb di *Newton.*

- **Installazione:** Rappresenta la facilità con cui è possibile installare il software. *ODE* richiede semplicemente di decomprimere i file presenti nell'archivio scaricato in una cartella di destinazione a nostra scelta. Ancora più semplicemente l'installazione di *Newton Game Dynamics* avviene attraverso un *installer* che ci permette di selezionare le opzioni principali.
- **Facilità di programmazione:** Indica la facilità con cui è possibile scrivere codice che utilizzi le funzionalità del programma. Entrambi i software sono abbastanza facili da programmare se si hanno delle conoscenze basilari di fisica e dei motori fisici, tuttavia *Newton Game Dynamics* richiede in generale meno passi per produrre una simulazione accettabile.
- **Facilità d'integrazione:** Indica quanto è facile utilizzare i due motori all'interno di altre applicazioni. Da questo punto di vista i software si equivalgono in quanto si devono compiere sostanzialmente le stesse operazioni per arrivare all'integrazione con il motore grafico e con il resto dell'applicazione: inclusione dell'header specifico del motore fisico, istruzioni per il *linker* su dove può trovare le librerie del motore fisico, creazione della simulazione fisica con tutti i suoi oggetti, mappatura di ogni oggetto fisico su un oggetto grafico.
- **Facilità di manutenzione:** Indica quanto è facile andare a modificare un software esistente per cambiare le impostazioni del motore fisico al fine di migliorarne le prestazioni o di risolvere dei problemi. Programmando la propria applicazione con un certo criterio è abbastanza facile tenere distinti gli aspetti fisici della simulazione dal resto del software per entrambi i motori fisici; entrambi i software, infatti, necessitano di poche istruzioni per interagire con il resto dell'applicazione.
- **Deployment:** Indica quanto è facile la messa in opera di software che sfruttano i due motori fisici. Per entrambi i motori è sufficiente copiare nella directory del programma principale una DLL contenente il codice del motore.

4.3.4 - Qualità

Il parametro "qualità" risponde a domande del tipo "quanti errori sono presenti nel codice?", "che grado di qualità raggiunge il codice scritto?". Non potendo rispondere in modo diretto a queste domande spesso si può giungere a delle valutazioni ponendoci delle domande che ci permettono di inferire i dati richiesti come:"quanti rilasci di minore entità sono stati rilasciati negli ultimi dodici mesi?", "quante patch sono uscite negli ultimi dodici mesi?". *ODE* negli ultimi dodici mesi ha visto

uscire la versone 0.8 e la versione 0.9; entrambe forniscono nuove funzionalità e correggono diversi bug. *Newton Game Dynamics* al contrario ha visto uscire l'ultimo aggiornamento nel maggio 2006.

Qualità	Open Dynamics Engine	Newton Game Dynamics
Rilasci minori negli ultimi 12 mesi	***	*
Patch uscite negli ultimi 12 mesi	***	*

Altri possibili metriche di valutazione della qualità possono riguardare il numero di bug risolti negli ultimi 6 mesi oppure quelli trovati ma non ancora risolti. Purtroppo queste metriche sono disponibili solo per *ODE* che presenta un proprio spazio su *SourceForge.Net*, mentre non sono disponibili per *Newton Game Dynamics* che è un prodotto *freeware* ma non Open Source.

4.3.5 - Prestazioni

Le prestazioni dei due motori fisici sono state valutate in due modi. Il primo riguarda le prestazioni dal punto di vista della velocità pura; questo viene fatto calcolando la media degli FPS (*Frame Per Second*) forniti dal motore grafico durante la simulazione. I risultati di queste rilevazioni sono mostrati nel grafico di figura 33. Il secondo metodo valuta le prestazioni dei due software riguardo al consumo di risorse di sistema, in particolare di memoria e CPU; questa valutazione è stata eseguita utilizzando il *task manager* di Windows e monitorando le applicazioni. In tabella sono riportati i risultati delle valutazioni mentre in figura 33 viene mostrato il grafico che evidenzia l'andamento degli FPS durante le 4 simulazioni eseguite.

Aspetti prestazionali	Open Dynamics Engine	Newton Game Dynamics
FPS	****	**
Uso CPU	**	**
Uso memoria	***	**

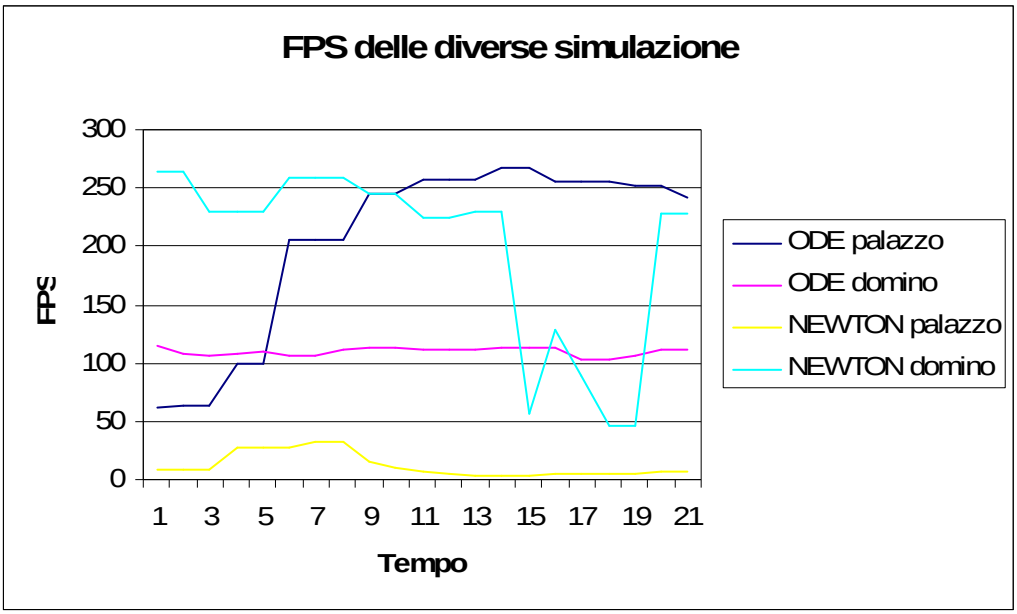

Figura 33 - Andamento degli FPS durante le simulazioni

Le prove effettuate evidenziano che *ODE* è generalmente più veloce di *Newton Game Dynamics* come numero di frame per secondo; inoltre *ODE* è anche più stabile nell'andamento mentre *Newton Game Dynamics* presenta molti "alti e bassi". Il consumo di CPU è invece equivalente in quanto entrambi i software utilizzano la CPU al 100% durante entrambe le simulazioni. Il consumo di memoria torna invece a essere a favore di *ODE* che in entrambe le simulazioni richiede all'incirca 3Mb in meno rispetto all'altro contendente.

4.3.6 - Scalabilità

La scalabilità indica quanto velocemente si deteriorano le prestazioni del software all'aumentare delle dimensioni del sistema. Per verificare questo aspetto sono state effettuate diverse prove sulle simulazioni del domino realizzate con i due motori fisici. In particolare si è scelto di partire da un minimo di 20 tessere nel percorso fino ad un massimo di 500 utilizzando un incremento di 20 tessere alla volta. Per ciascuna configurazione è stato misurato il tempo che intercorre dall'applicazione della forza sulla prima tessera fino alla caduta dell'ultima. I risultati di queste prove sono visibili nel grafico di figura 34 da cui si evince chiaramente che, sebbene *Open Dynamics Engine* si dimostri nel complesso sempre più veloce di *Newton Game Dynamics*, quest'ultimo ha un comportamento più simile a una funzione lineare ed è quindi preferibile dal punto di vista della scalabilità. Prove analoghe sono state eseguite anche sulla simulazione del

palazzo ma hanno fornito risultati meno attendibili a causa del diverso comportamento fisico dei due motori.

Scalabilità	Open Dynamics Engine	Newton Game Dynamics
Valutazione sulla scalabilità del sistema	***	****

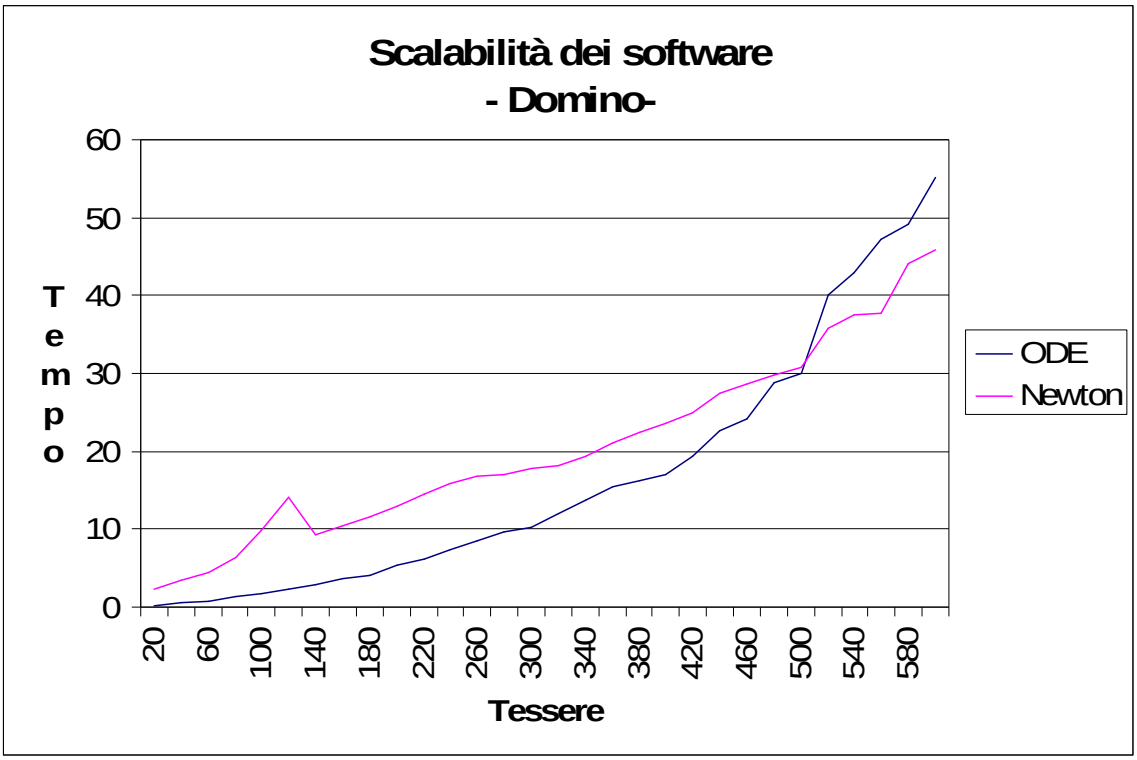

Figura 34 - Prestazioni dei software al crescere delle dimensioni del sistema

4.3.7 - Architettura

La valutazione dell'architettura dei software ha come obiettivo il dare una risposta a domande simili: "è possibile utilizzare *plugins* di terze parti?", "il software fornisce *API* esterne?", "esiste la possibilità di attivare o deattivare alcune caratteristiche a *runtime*?"

Architettura	Open Dynamics Engine	Newton Game Dynamics
Plugins esterni	**	**
Utilizzo di API	*****	*****
Caratteristiche modificabili a runtime	*****	*****

Da questo punto di vista i software si equivalgono; entrambi infatti permettono di utilizzare vincoli creati dal programmatore e funzioni di *callback* personalizzate per le collisioni; inoltre essendo due librerie il loro uso è basato esclusivamente sull'uso di API e sull'attivazione delle caratteristiche via codice a runtime.

4.3.8 - Supporto

Valuta il grado di supporto fornito per l'utilizzo del software. Le metriche che si possono utilizzare per valutare questo aspetto comprendono il volume di messaggi mensili della mailing list del software e la disponibilità o meno di un servizio di help professionale. Nessuno dei due software in esame mette a disposizione servizi di help professionale; *ODE* mette a disposizione una mailing list con un volume di messaggi che oscilla intorno ai 150 mensili. *Newton Game Dynamics* invece propone un forum ufficiale in cui si offre supporto agli utlizzatori.

Supporto	Open Dynamics Engine	Newton Game Dynamics
Help professionale	*	*
Volume di messaggi della mailing list	***	*
Altre forme di supporto	*	**

4.3.9 - Documentazione

Indica il grado di documentazione fornita assieme al software. Entrambi i software hanno un sito web basato su *Wiki* che mette a disposizione una buona documentazione. Nei file che vengono scaricati con il download *ODE* fornisce una serie di documenti *HTML* che parlano principalmente della struttura dei file *C* del motore, al contrario *Newton Game Dynamics* propone una serie di tutorial divisi per argomento e un indice *HTML* delle *API*. Tuttavia *ODE* mette a disposizione una guida in formato PDF molto accurata anche se priva di esempi pratici.

Documentazione	Open Dynamics Engine	Newton Game Dynamics
Materiale disponibile	****	****

4.3.10 - Adozione

Indica quanto il software è stato adottato dalle comunità dei programmatori. Le due metriche utilizzate per la valutazione di questo parametro sono il numero di libri dedicati al software che sono stati pubblicati e il numero di progetti in cui il software è stato utilizzato. Per quel che riguarda il numero di libri dedicati si è proceduto con una ricerca su *amazon.com*. La ricerca ha trovato un solo titolo disponibile per *ODE* (peraltro citato sullo stesso sito ufficiale) e nessuno per *Newton Game Dynamics*. Per il conteggio dei progetti nei quali il software viene utilizzato invece si è deciso di conteggiare quelli indicati dai rispettivi siti ufficiali: i risultati evidenziano 99 progetti in cui è stato utilizzato *Open Dynamics Engine* contro i 53 che usano *Newton Game Dynamics*.

Adozione	Open Dynamics Engine	Newton Game Dynamics
Libri dedicati	**	*
Numero di progetti	****	**

4.3.11 - Community

Rappresenta il grado di attività della comunità di sviluppo che ruota attorno al software. Le metriche che si possono utilizzare per la valutazione riguardano il volume di messaggi delle mailing list e il numero di programmatori che hanno in qualche modo contribuito al progetto. Per la nostra valutazione questo parametro non è stato preso in considerazione.

4.3.12 - Professionalità

Indica se il progetto è sviluppato da semplici amatori oppure da organizzazioni indipendenti o ancora se dietro il team di sviluppo ci sono grandi aziende che finanziano il progetto. Per la nostra valutazione questo parametro non è stato preso in considerazione.

4.3.13 - Sicurezza

Indica il grado di sicurezza del software. Le possibili metriche per la valutazione di questo parametro sono il numero di falle di sicurezza risolte negli ultimi sei mesi, il numero di falle di sicurezza ancora aperte e la presenza o meno di documentazione esplicitamente dedicata alla sicurezza. Per la nostra valutazione questo parametro non è stato preso in considerazione.

4.4 - Traduzione dei dati

In questa fase utilizziamo i pesi che abbiamo scelto nella seconda fase sulle valutazioni che abbiamo fatto per ottenere dei dati numerici. La formula generale è data da:

*PesoDellaMetrica * Votazione*

Poi i valori di tutte le metriche vengono sommati generando una valutazione complessiva della categoria.

Funzionalità		**ODE**		**NEWTON**	
	Peso	Voto		Voto	
Verosimiglianza della simulazione	0,2	3	0,6	4	0,8
Rilevamento collisioni	0,1	4	0,4	4	0,4
Creazione geometrie semplici	0,1	3	0,3	3	0,3
Gestione collisioni	0,1	2	0,2	4	0,4
Distribuzione di massa	0,1	3	0,3	4	0,4
Resa attrito	0,1	2	0,2	4	0,4
Applicazione forze	0,05	4	0,2	2	0,1
Integratori	0,05	3	0,15	4	0,2
Geometrie complesse	0,05	3	0,15	3	0,15
Vincoli	0,05	3	0,15	3	0,15
Supporto	0,05	4	0,2	3	0,15
Ottimizzazioni	0,05	4	0,2	3	0,15
	1		**3,05**		**3,6**

Prestazioni		**ODE**		**NEWTON**	
	Peso	Voto		Voto	
FPS	0,4	4	1,6	2	0,8
uso CPU	0,3	2	0,6	2	0,6
uso memoria	0,3	3	0,9	2	0,6
	1		**3,1**		**2**

Scalabilità		**ODE**		**NEWTON**	
	Peso	Voto		Voto	
Valutazione generale	**1**	3	**3**	4	**4**

Usabilità		**ODE**		**NEWTON**	
	Peso	Voto		Voto	
Facilità programmazione	0,2	3	0,6	4	0,8
Reperimento	0,15	4	0,6	5	0,75
Integrazione	0,15	4	0,6	4	0,6
Deployment	0,15	5	0,75	5	0,75
Manutenzione	0,15	4	0,6	4	0,6
Download	0,1	5	0,5	3	0,3
Installazione	0,1	4	0,4	5	0,5
	1		**4,05**		**4,3**

Qualità		**ODE**		**NEWTON**	
	Peso	Voto		Voto	
Rilasci ultimi 12 mesi	0,5	3	1,5	1	0,5
Patch ultimi 12 mesi	0,5	3	1,5	1	0,5
	1		**3**		**1**

Architettura		**ODE**		**NEWTON**	
	Peso	Voto		Voto	
Modifiche a runtime	0,4	5	2	5	2
Uso API	0,4	5	2	5	2
Uso Plugin	0,2	2	0,4	2	0,4
	1		**4,4**		**4,4**

Supporto		**ODE**		**NEWTON**	
	Peso	Voto		Voto	
Help professionale	0,4	1	0,4	1	0,4
Volume messaggi mailing list	0,3	3	0,9	1	0,3
Altre forme di supporto	0,3	1	0,3	2	0,6
	1		**1,6**		**1,3**

Documentazione		**ODE**		**NEWTON**	
	Peso	Voto		Voto	
Materiale disponibile	**1**	4	**4**	4	**4**

Adozione		**ODE**		**NEWTON**	
	Peso	Voto		Voto	
Numero di progetti	0,7	4	2,8	2	1,4
Numero di libri dedicati	0,3	2	0,6	1	0,3
	1		**3,4**		**1,7**

Nella seconda fase sommiamo i totali delle singole categoria pesandoli con i pesi scelti in precedenza per le categorie stesse. Ecco la tabella che mostra i risultati di questa fase:

Categoria		**ODE**		**NEWTON**	
	Peso	Voto totale		Voto totale	
Funzionalità	0,25	3,05	0,7625	3,6	0,9
Prestazioni	0,2	3,1	0,62	2	0,4
Scalabilità	0,15	3	0,45	4	0,6
Usabilità	0,1	4,05	0,405	4,3	0,43
Qualità	0,1	3	0,3	1	0,1
Architettura	0,05	4,4	0,22	4,4	0,22
Supporto	0,05	1,6	0,08	1,3	0,065
Documentazione	0,05	4	0,2	4	0,2
Adozione	0,05	3,4	0,17	1,7	0,085
Community	0	0	0	0	0
Professionalità	0	0	0	0	0
Sicurezza	0	0	0	0	0
	1		**3,2075**		**3**

Conclusioni

Nel capitolo quattro abbiamo visto l'applicazione completa del modello BRR. I risultati riportati nell'ultima tabella riportano un risultato di **3,2075** per *Open Dynamics Engine* a confronto del **3** ottenuto da *Newton game Dynamics* quindi, anche se di poco, il vincitore della nostra sfida risulta essere *ODE*; Se ci trovassimo in un ambito aziendale tale risultato starebbe a significare che *ODE* si è rivelato un software più completo e maturo ***per i nostri scopi***. È importante sottolineare soprattutto l'ultima frase in quanto il modello BRR è concepito proprio per formulare un giudizio valido nel rispetto dell'orientamento operativo del software che stiamo valutando.

Nel nostro caso è giusto ricordare che stavamo cercando un motore fisico *real time* da utilizzare all'interno di un videogame; proprio per questo nell'applicazione del modello abbiamo dato la priorità ad aspetti come le funzionalità offerte dall'engine (in modo particolare alla verosimiglianza globale della simulazione), alle prestazioni ottenute durante le prove sulle due simulazioni e alla scalabilità del sistema (riportiamo nuovamente in figura 37 e 38 i risultati di tali prove).

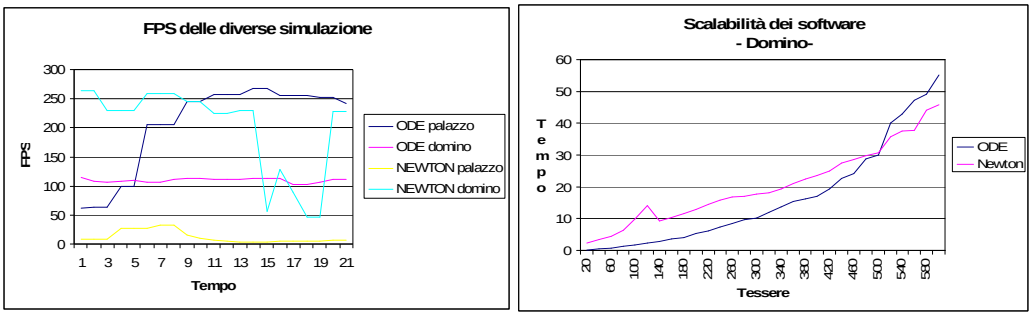

Figura 35 – Grafici delle prestazioni e della scalabilità dei due software

La somma delle prime tre categorie considerate per *Newton Game Dynamics* da un parziale di **1,9** contro l'**1,8325** ottenuto da ODE: se ci fermassimo a questa analisi superficiale avremmo potuto scegliere *Newton Game Dynamics* come software da adottare per le nostre simulazioni, invece l'applicazione del modello BRR ci fornisce una visione più generale delle caratteristiche dei

software che evidenzia come *Newton Game Dynamics* sia carente in fatto di documentazione e adozione e come i rilasci di versioni aggiornate e corrette siano molto più ridotti rispetto a ODE.

Inoltre proprio per la sua natura di prodotto Open Source ODE risulta essere superiore in fatto di supporto e qualità del software, al contrario di *Newton Game Dynamics* che è un prodotto free ma il cui codice sorgente non è disponibile: correzione dei bug e inserimento di nuove funzionalità sono competenza dei soli sviluppatori che non possono appoggiarsi al contributo dei numerosi utenti che solitamente contribuiscono allo sviluppo dei prodotti Open Source.

Bibliografia

- Wikipedia [autori vari]
- Fisica: principi e applicazioni [Giancoli]
- Open Dynamics Engine: user guide [Russell Smith]
- ODE for newbies [David Coombes]
- Open Dynamics Engine: corso di realtà virtuale [Christian Bianchi]
- Using ODE with Irrlicht [Thomas Suter]
- NewtonDynamics.com []
- Newton Game Dynamics API [Julio Jerez]
- Integrating Newton Game Dynamics with Irrlicht []
- Unofficial Newton Game Dynamics Wiki []
- WWW.OpenBRR.org []
- Business Readiness Rating for Open Source []