UNIVERSITÀ DI PISA

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI
CORSO DI LAUREA SPECIALISTICA IN TECNOLOGIE INFORMATICHE

TESI DI LAUREA

PROGETTAZIONE E SVILUPPO DI UN FRAMEWORK PER LA DEFINIZIONE E L'UTILIZZO DI METAFORE DI INTERAZIONE PER AMBIENTI VIRTUALI

RELATORI
Dott. Ing. Marcello CARROZZINO
Dott. Ing. Franco TECCHIA

CANDIDATO
Andrea SALVADORI

SESSIONE DI LAUREA 12 OTTOBRE 2012
ANNO ACCADEMICO 2011/12

Indice

Introduzione

Dall'inizio degli anni '60 sono stati compiuti sforzi notevoli nella ricerca e nello sviluppo di sistemi interattivi. La prima fondamentale tappa di questo processo viene generalmente considerata la realizzazione di *Sketchpad*, un sistema sviluppato da *Ivan Sutherland* nel 1963 [36] che consentiva all'utente di manipolare oggetti grafici mediante una penna ottica. Questo sistema ha gettato le basi per le successive ricerche sulle interfacce uomo-macchina, che si sono concretizzate negli anni '70 con la creazione dei primi sistemi dotati di interfaccia grafica, ad opera di *Alan Kay* e del suo gruppo di ricerca. La metafora del desktop e l'uso del mouse come dispositivo di puntamento sono ormai diventati uno standard de facto, grazie alla sua adozione nei sistemi Macintosh e Windows.

Nel campo della realtà virtuale la varietà di dispositivi è sempre stata nettamente più ricca: basti pensare ai molti sistemi di tracking, alle interfacce aptiche ecc., che unitamente ad hardware grafico di alto livello e a dispositivi di visualizzazione stereoscopica, consentono la realizzazione di sistemi ad elevata interattività ed immersività. Tuttavia, le modalità con le quali l'utente interagisce con l'ambiente virtuale vengono troppo spesso definite attraverso soluzioni ad hoc. Buona parte della ricerca in questo settore è stata quindi orientata alla definizione di un insieme di modelli concettuali che astraggono le modalità con le quali viene svolto un determinato compito all'interno dell'ambiente virtuale. Tali modelli sono noti come di *metafore di interazione*.

Il progetto di tesi consiste nella progettazione e nella realizzazione di un framework per semplificare l'implementazione dell'interazione utente in applicazioni tridimensionali interattive e di realtà virtuale. Il framework è stato sviluppato in XVR, un ambiente integrato per la realizzazione di applicazioni di realtà virtuale, sviluppato presso il laboratorio PERCRO della Scuola Superiore Sant'Anna di Pisa.

Esso definisce un insieme di metafore di interazione già pronte all'uso, e fornisce al programmatore gli strumenti necessari per crearne di nuove. Le metafore presenti sono state scelte tra quelle più usate nell'ambito applicativo della fruizione di contenuti culturali e supportano funzionalità quali *collision detection* e *terrain following* della camera. A tal fine è stata definita una libreria contenente gli algoritmi e le strutture dati necessarie all'esecuzione di alcuni tipi di test di intersezione tra primitive geometriche.

Uno degli obiettivi principali che il framework si propone è quello di rendere le metafore e le altre funzionalità offerte indipendenti dal tipo di dispositivo usato. Inoltre l'architettura adottata consente di uniformare e semplificare la gestione dell'input rispetto a quanto avviene nei programmi XVR tradizionali.

Allo stato attuale i dispositivi di input supportati sono il mouse, la tastiera ed il touch-screen. In via sperimentale è presente anche il supporto al Microsoft Kinect, quale esempio di dispositivo di tracking. Viene comunque data la possibilità aggiungere il supporto ad ulteriori dispositivi, anche di natura diversa.

Infine è stata definita una libreria minimale per la realizzazione di semplici interfacce grafiche bidimensionali. La libreria è progettata in modo da poter essere impiegata con dispositivi di diversa natura.

Sommario

La tesi è strutturata in tre parti. Nella prima viene fornita un'introduzione alle interfacce utente e alla realtà virtuale, e vengono introdotti i concetti di *metafora di interazione* e *tecnica di interazione*. Vengono inoltre presentate alcune delle metafore più note o innovative presenti in letteratura.

La seconda parte tratta il framework sviluppato per questa tesi. In particolare, il capitolo 2 fornisce una panoramica dell'architettura e delle caratteristiche del framework. Vengono quindi presentate le principali idee che ne hanno guidato lo sviluppo, oltre che alcuni esempi sul suo utilizzo. Nei successivi capitoli sono descritti nel dettaglio i vari moduli che lo, sia dal punto di vista implementativo sia in riferimento al loro utilizzo.

L'ultima parte è dedicata ai risultati sperimentali ed ai test di usabilità.

Parte I

Ambienti virtuali e metafore di interazione

Capitolo 1

Metafore di interazione

1.1 La Realtà Virtuale

Il termine *"Virtual Reality"* è stato introdotto per la prima volta alla fine degli anni '80 da *Jaron Lanier*, fondatore della *Virtual Programming Languages Reaserch*. Durante un'intervista per la rivista *Whole Earth Review*, *Lanier* definì la realtà virtuale

> "una tecnologia che usa vestiario computerizzato per sintetizzare la realtà"

Una migliore definizione viene data in [14]:

> "La realtà virtuale è un'esperienza sensoriale generata da un computer che permette all'utente di immedesimarsi in essa, riuscendo a stento a distinguere l'esperienza "virtuale" da quella reale"

La qualità dell'esperienza virtuale dipende da tre fattori:

Presenza Il coinvolgimento mentale dell'utente nell'esperienza virtuale. Dipende dalla predisposizione dell'utente a immedesimarsi nella simulazione.

Immersività Il coinvolgimento fisico, indotto dagli stimoli sensoriali. Consente all'utente di percepire l'ambiente sintetico come reale.

Interazione La possibilità che ha l'utente di interagire con l'ambiente virtuale per mezzo delle proprie azioni, ricevendo un feedback in tempo reale sulle modifiche apportate all'ambiente. Fornisce una misura del realismo della simulazione.

Possiamo classificare i sistemi di realtà virtuale in tre categorie, ognuna delle quali ha caratteristiche diverse in termini di *presenza*, *immersività* e *interattività*: VR Testuale, VR Desktop e VR Immersiva.

Nei sistemi *testuali* l'ambiente viene rappresentato tramite descrizioni testuali e non grafiche. Questi sistemi, nati in ambito videoludico, possono produrre un forte senso di presenza nonostante la mancanza di immersività.

Quella *immersiva* è la realtà virtuale propriamente detta, nella quale le tecnologie utilizzate garantiscono un alto livello di immersività, fornendo all'utente un'esperienza sensoriale tale da farlo sentire parte integrante del mondo virtuale. I sistemi di questo tipo sono tipicamente dotati di visione stereoscopica, dispositivi per il tracking della posizione e dei movimenti dell'utente, feedback auditivo ed (eventualmente) aptico ecc.

Poiché i dispositivi impiegati nei sistemi immersivi sono tipicamente ingombranti e molto costosi (e quindi a disposizione di pochi laboratori specializzati), si è dovuto considerare come sistemi di realtà virtuale anche quelli da "*scrivania*". Qui l'interazione avviene per mezzo di dispositivi economici e largamente diffusi come mouse, tastiera, joystick ecc. Anche se questo riduce l'immersività della simulazione, per molte persone rappresenta l'unica possibilità di avere accesso alla realtà virtuale.

Il laboratorio PERCRO[1] della Scuola Superiore S. Anna di Pisa opera nell'ambito della realtà virtuale, con particolare riferimento alle tematiche di:

- Sviluppo di sistemi di interfaccia tattile e per il ritorno di forza.
- Sviluppo di rendering grafico ad alta definizione in tempo reale.
- Studio delle problematiche relative alla modellazione.

1.2 Le interfacce utente

Per **interfaccia utente** (UI) si intende il mezzo attraverso cui l'utente interagisce con un computer, e comprende sia i dispositivi fisici di input e output, sia le componenti software che interpretano l'input dell'utente.

Dall'introduzione dei computer desktop negli anni '70, sono stati compiuti sforzi notevoli nella ricerca e nello sviluppo di interfacce utente per personal computer. Di fondamentale importanza sono state le ricerche di Alan Kay al Xerox Palo Alto Research Center nei primi anni '70, che hanno portato alla creazione dei primi sistemi con *interfaccia grafica* (GUI), del *mouse* come nuovo dispositivo di input da affiancare alla tastiera, e della *metafora del desktop*. Questa interfaccia si è diffusa rapidamente (grazie alla sua adozione nei sistemi *Macintosh* e *Windows*) e, nonostante i molti miglioramenti introdotti in questi ultimi decenni, la sua essenza è rimasta la stessa di allora. Solo ultimamente

[1] Il PERCeption RObotics è un laboratorio di ricerca che ha come missione lo sviluppo di nuovi sistemi e tecnologie nell'ambito della Realtà Virtuale e Tele-Robotica.

stiamo assistendo ad una evoluzione, trainata dalla diffusione di *superfici multi-touch*. Adottate inizialmente in ambito mobile per sopperire alla mancanza di un dispositivo di puntamento, le superfici multi-touch stanno gradualmente influenzando lo sviluppo delle interfacce utente anche in ambito desktop.

Nel seguito di questa tesi ci occuperemo delle interfacce utente per *ambienti virtuali tridimensionali* (*Virtual Environments* - VE). Anche questo settore ha avuto un'incredibile evoluzione negli ultimi decenni: negli anni '80 sono comparse sul mercato le prime piattaforme a basso costo orientate o dedicate al gaming, che hanno portato al successo del mercato videoludico. Tale successo ha stimolato la ricerca nel campo della computer graphics, sia in ambito algoritmico che in quello dell'hardware grafico, permettendo la realizzazione di ambienti virtuali sempre più realistici. Tuttavia, fino a qualche anno fa, i dispositivi di input in ambito consumer sono rimasti pressoché gli stessi: mouse e tastiera su PC, e joystick o gamepad su console. Nel campo della realtà virtuale la varietà di dispositivi a disposizione è sempre stata nettamente più ricca: basti pensare ai molti sistemi di tracking, alle interfacce aptiche ecc., che unitamente ad hardware grafico di alto livello e a dispositivi di visualizzazione stereoscopica quali head-mounted display, CAVE ecc., consentono di implementare simulazioni ben più realistiche ed immersive. Questi dispositivi sono però generalmente costosi e spesso ingombranti, fattori che ne hanno limitato la diffusione a pochi laboratori di ricerca. Questa situazione, nel suo complesso, ha di fatto contenuto l'evoluzione delle interfacce utente per ambienti virtuali immersivi. Un punto di svolta si è avuto negli ultimi anni grazie al mercato delle console, dove si è cercato di dare nuova linfa al settore puntando sull'immersività. L'idea chiave è stata quella di riprogettare alcuni dispositivi tipici della realtà virtuale al fine di ottenerne delle varianti di costo e dimensioni contenuti. Esempi significativi sono il Kinect della Microsoft e il Wii Remote della Nintendo. È quindi ragionevole prevedere che la diffusione di questi dispositivi agisca da stimolo per nuove ricerche sulle interfacce utente.

Un aspetto chiave nella realizzazione di una UI consiste nello stabilire come l'input dell'utente deve essere tradotto in azioni all'interno del VE. È qui che entra in gioco il concetto di **metafora**.

1.3 Le metafore e il loro ruolo

Secondo il dizionario della lingua italiana[2], una metafora è una

> "figura retorica consistente nel trasferire un termine dal suo significato proprio a uno figurato, secondo un rapporto analogico"

[2]Grande Dizionario Hoepli di Gabrielli Aldo

Da questa definizione possiamo quindi cogliere un trasferimento di significato, ovvero di conoscenza, secondo un rapporto di analogia. In altri termini, lo scopo di una metafora è quello di

> "imitare i concetti già noti all'utente in un altro contesto, in modo da trasferire questa conoscenza ad un nuovo compito nel nuovo contesto" [3]

Una definizione ancora più specifica nell'ambito della *Human-Computer Interaction* (HCI) è la seguente:

> "La metafora di interazione fornisce all'utente un *modello* che gli consente di *prevedere il funzionamento del sistema* in conseguenza a diversi tipi di input" [40]

Una buona metafora dovrebbe essere semplice da imparare (sfruttando appunto le conoscenze precedenti), ma anche efficace e semplice da usare. Infatti non sempre la metafora che risulta più naturale (in analogia al mondo reale) risulta la migliore in termini di usabilità. Come esempio consideriamo la metafora del *desktop*. Essa è una buona metafora perché sfrutta nell'ambito delle interfacce grafiche alcuni concetti tipici del lavoro di ufficio[3], quali scrivania, documenti, cartelle, cestino ecc. Questo ha facilitato l'apprendimento e l'uso del calcolatore, consentendo all'utente di operare con concetti già noti, anche se trasposti in un ambito differente.

Fino ad ora abbiamo esaminato due aspetti dell'interfaccia utente: i dispositivi fisici e le metafore. Resta però da stabilire come l'input dell'utente venga tradotto in azioni all'interno del VE, in accordo al modello concettuale ed i vincoli definiti dalla metafora. È questo il compito delle **tecniche di interazione** [22, 25]. Ovviamente occorre definire una differente tecnica di interazione per ogni coppia <dispositivo di input, metafora>, tuttavia per ognuna di tali coppie possono esistere molte tecniche di interazione diverse. Segue che non sempre l'applicabilità di una metafora è vincolata dall'utilizzo di specifici dispositivi[4], in quanto l'usabilità complessiva dipende fortemente da come viene definita la tecnica di interazione che lega i due.

Riassumiamo quanto detto fino ad ora:
Per *interfaccia utente* si intende il mezzo attraverso cui l'utente interagisce con un computer. Essa è costituita da:

- **Dispositivi** di input ed output che consentono all'utente di interagire fisicamente con il sistema.

[3] Uno dei settori in cui i primi personal computer erano maggiormente diffusi.
[4] Sebbene esistano metafore pensate per essere impiegate esclusivamente con determinati dispositivi.

- **Metafore**, che forniscono un modello che consente all'utente di prevedere il funzionamento del sistema. Ogni metafora definisce *modalità* e *vincoli* per lo svolgimento di un task, in accordo al modello.
- **Tecniche di interazione**, che traducono l'input dell'utente in azioni all'interno del VE, applicando una specifica metafora.

1.4 Classificazione delle metafore

Le metafore di interazione possono essere classificate secondo diversi criteri. La classificazione più significativa per i nostri scopi è quella che mette in relazione le metafore con i compiti comunemente svolti dagli utenti. Nel campo degli ambienti virtuali, tali *task* possono essere classificati in tre categorie [15]:

- **Navigazione** all'interno della scena, ovvero il controllo della posizione e dell'orientamento del punto di vista all'interno dell'ambiente virtuale.
- **Selezione** degli elementi della scena.
- **Manipolazione** di tali elementi.

Una qualsiasi applicazione non banale deve saper integrare metafore differenti, dipendenti dalle funzionalità messe a disposizione dall'applicazione stessa. Per ogni tipo di *task* possono anche coesistere metafore differenti: così facendo l'utente avrà la possibilità di svolgere compiti simili (se non gli stessi) con modalità differenti. È inoltre possibile combinare più metafore per ottenere metafore complesse.

Nei paragrafi successivi esamineremo alcune tra le più note o innovative metafore di interazione e, per ognuna di esse, descriveremo le principali tecniche di interazione.

1.5 Metafore per la navigazione

1.5.1 Flying vehicle

La metafora di navigazione più semplice e nota è la cosiddetta "**Flying vehicle**", attraverso la quale l'utente si sposta all'interno della scena controllando un veicolo volante virtuale sul quale è posta la camera [40]. Ovviamente questa metafora non necessita di una fisica del volo accurata, quindi il veicolo può restare "sospeso in aria" ed eseguire qualsiasi tipo di manovra. Inoltre, sebbene la metafora preveda per sua natura sei gradi

di libertà, spesso alcuni gradi di libertà non sono controllabili dall'utente, a causa dei vincoli posti dall'applicazione o al fine di rendere l'interazione più semplice[5]. Nel corso del tempo sono state proposte varie tecniche per il controllo del veicolo. Di seguito vengono descritte le principali, classificate in funzione della classe di dispositivi impiegati.

Tecniche di interazione in ambito desktop

Il controllo del veicolo virtuale avviene, nella maggior parte dei casi, utilizzando mouse, tastiera o joystick come input device. Per loro natura però, questi dispositivi permettono di controllare soltanto due gradi di libertà, contro i 5/6 offerti dalla metafora e tipicamente richiesti dalle applicazioni tridimensionali. I gradi di libertà restanti vengono perciò controllati (almeno in parte) sfruttando tasti/pulsanti aggiuntivi, oppure attraverso l'utilizzo combinato di due periferiche. L'esempio classico prevede l'impiego contemporaneo di mouse e tastiera: il mouse consente di modificare l'orientamento della camera, mentre i tasti direzionali della tastiera ne controllano lo spostamento.

Un approccio alternativo consiste nell'impiego di *gesture*, eseguite con un qualsiasi dispositivo di puntamento. Il principale difetto di queste tecniche è la loro difficoltà di apprendimento, che risulta inaccettabile in alcuni ambiti applicativi.

Tecniche di interazione in ambienti immersivi

Vediamo adesso alcune delle tecniche impiegabili in ambienti immersivi dove, grazie a dispositivi di tracking, l'utente può interagire direttamente con il VE sfruttando il proprio corpo come input device.

Queste tecniche differiscono tra loro in funzione di come vengono determinate velocità e direzione di spostamento [25]. Per quanto riguarda la direzione, una possibile classificazione è la seguente:

- **Hand-directed flying**: la direzione di spostamento è determinata dalla posizione e dall'orientamento della mano. È noto anche come *"Pointing"*, in quanto l'utente deve semplicemente "puntare" la mano nella direzione in cui intende spostarsi. Questa tecnica ha il vantaggio di non limitare la libertà di movimento della testa durante la navigazione, cosa che può però creare confusione negli utenti meno esperti.

[5] Tipicamente viene disabilitata la rotazione attorno all'asse longitudinale del veicolo (rollio).

Figura 1.1: Hand-directed flying

- **Two-handed flying**: è sostanzialmente una variante a due mani dell'*hand-directed flying*. In questo caso la direzione di volo è definita dal vettore che congiunge le due mani (il verso è dato dalla mano dominante). Tipicamente la velocità di spostamento viene calcolata in proporzione alla distanza tra le due mani.

- **Gaze-directed flying**: l'idea alla base di questa tecnica è quella di usare come direzione di spostamento la direzione di vista dell'utente. In assenza di dispositivi di *eye-tracking*, la direzione di vista può essere approssimata con l'orientamento della testa. Sebbene questa metafora risulti particolarmente intuitiva per gli utenti inesperti, comporta un continuo movimento della testa, cosa che può risultare stancante. Inoltre, a differenza dell'*hand-directed flying*, non consente di osservare liberamente la scena durante la navigazione.

- **Crosshair**: combina l'*hand-directed* con il *gaze-directed* flying. La direzione di spostamento è determinata dal vettore che, partendo dall'occhio dominante dell'utente, passa attraverso un oggetto di puntamento da lui impugnato (può essere anche un dito della mano la cui posizione è rilevata tramite tracking). L'utente può quindi spostarsi verso un oggetto semplicemente mirandolo.

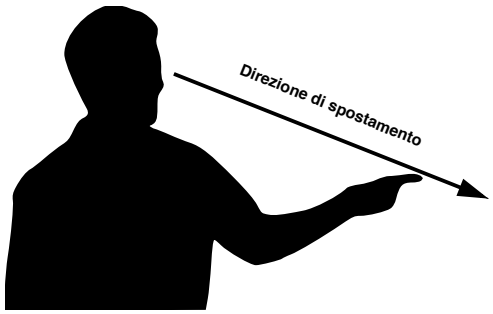


Figura 1.2: Crosshair

Passiamo ora alla determinazione della velocità. Le tecniche più comuni sono le seguenti:

- **Velocità costante**: è la soluzione più semplice. Tale velocità può essere scelta arbitrariamente o essere proporzionale alla dimensione della scena, in modo da per-

correre l'intera scena in tempi ragionevoli. Questa tecnica tende tuttavia a produrre movimenti frammentari ed imprecisi.

- **Accelerazione costante**: con questo approccio si ottiene una bassa velocità all'inizio della navigazione, consentendo così spostamenti brevi e precisi, per poi crescere linearmente col passare del tempo, in modo da coprire ampie distanze in tempi brevi. In genere lo spostamento si conclude con una fase a decelerazione costante, che rende l'arresto più morbido. La principale difficoltà nell'impiego di questa tecnica consiste nella determinazione di un buon valore di accelerazione.

- **Hand-controlled**: la velocità viene determinata in funzione della distanza tra la mano ed il busto dell'utente. Nel caso del *two-handed flying* la velocità sarà invece proporzionale alla distanza tra le due mani. Funzioni tipiche che associano la velocità alla distanza della mano sono quelle lineari o esponenziali. In alternativa è possibile utilizzare una funzione a tratti che suddivide logicamente l'area di estensione del braccio in tre regioni: quando la mano si trova nella prima regione si ha un moto ad accelerazione costante, nella seconda si mantiene invariata la velocità, mentre nella terza si ha una decelerazione costante. Funzioni simili consentono di ottenere un buon controllo della velocità unito ad un ampio range di velocità impiegabili. La principale debolezza di questa tecnica consiste nel fatto che l'utente è costretto a mantenere il braccio disteso per periodi di tempo relativamente lunghi, cosa che può risultare stancante.

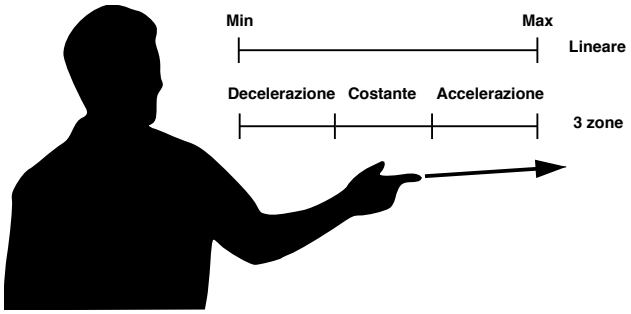


Figura 1.3: Hand-controlled speed

Tecniche di interazione per interfacce aptiche

Le interfacce aptiche sono speciali dispositivi di input capaci di restituire un feedback tattile all'utente, cosa che aumenta notevolmente l'usabilità e l'immersività dell'applicazione. Grande successo hanno avuto le interfacce aptiche desktop come il Sensable Phantom[6]. In [7] viene proposta una tecnica di interazione per controllare il flying vehicle con questo tipo di dispositivi. L'idea è quella di racchiudere la stylus del Phantom all'interno di un

[6] http://www.sensable.com

box virtuale. L'utente può spostare e ruotare liberamente la stylus all'interno del box, riuscendo al contempo a percepirne i bordi come se si trattasse di un oggetto fisico reale. L'orientamento del veicolo viene controllato semplicemente orientando la stylus nella direzione desiderata. Quando però l'utente preme la stylus su di una faccia del box, il veicolo inizia a muoversi nella direzione associata alla faccia premuta. Inoltre tali facce vengono percepite dall'utente come se fossero costituite da materiale elastico. Così facendo l'utente può regolare la velocità del veicolo aumentando o diminuendo la pressione esercitata sulle pareti.

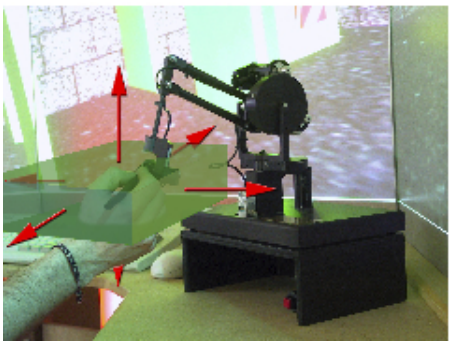


Figura 1.4: Utilizzo di un Sensable Phantom per controllare la metafora flying vehicle

1.5.2 Walking

Come si intuisce dal nome, l'utente naviga all'interno dell'ambiente virtuale semplicemente camminando in esso. Di fatto si può parlare di *walking* anche quando viene simulata la guida di un veicolo terrestre. Rispetto al flying vehicle l'utente ha a disposizione un grado di libertà in meno per spostarsi nell'ambiente.

Tecniche di interazione

Tutte le tecniche viste per il *flying vehicle* possono essere utilizzate anche con questa metafora, con qualche accorgimento per tener conto del grado di libertà mancante. Oltre a queste sono state sviluppate tecniche specifiche per il *walking*. Nel caso di ambienti immersivi, uno degli approcci più popolari prevede che l'utente si muova all'interno del VE nella maniera più diretta e naturale possibile: camminando. La scena tridimensionale è quindi "compresente" con la stanza reale e l'utente si muove nella scena camminando all'interno della stanza reale. La visualizzazione della scena può avvenire per mezzo di un *Head-Mounted Display* (HMD) oppure all'interno di un *CAVE*. In entrambi i casi è necessario un meccanismo di tracking per rilevare posizione e orientamento della testa dell'utente[7]. Questa tecnica presenta però due problemi:

[7]Il dispositivo di tracking può essere integrato direttamente nell'HMD

- La porzione di scena visitabile è limitata dalle dimensioni della stanza reale.
- La velocità di spostamento può risultare troppo bassa per scene di grandi dimensioni.

Il primo problema può essere risolto completamente utilizzando un treadmill. Per ovviare al secondo (e per attenuare il primo se non si dispone di un treadmill) si può fornire una modalità di navigazione aggiuntiva, come il "**Seven League Boots**" [19]. Quando l'utente attiva questa modalità, ogni suo passo nel mondo reale produrrà uno spostamento amplificato all'interno dell'ambiente virtuale. Ad esempio ogni passo nel mondo reale può corrispondere a sette passi all'interno della scena. Si noti però che non basta moltiplicare ogni vettore spostamento per un fattore di scala. Infatti, camminando, l'utente produce anche delle oscillazioni orizzontali. Se ci limitassimo ad applicare un fattore di scala, anche tali oscillazioni verrebbero amplificate, disorientando l'utente. La soluzione corretta consiste nello scomporre il vettore spostamento $\vec{D}$ come combinazione lineare di due versori $\vec{V}$ e $\vec{W}$, che rappresentano rispettivamente la direzione di spostamento prevista e la direzione di oscillazione[8]:

$$\vec{D} = d_V \cdot \vec{V} + d_W \cdot \vec{W}$$

A questo punto basta moltiplicare per il fattore di scala la sola componente d_V. La direzione di spostamento $\vec{V}$ si ricava dall'orientamento della testa; si parla quindi di *gaze-directed walking* (vedi pagina 10). Per consentire all'utente una certa libertà di movimento della testa senza però che la direzione cambi continuamente, $\vec{V}$ viene calcolato come la media pesata tra la direzione attuale di vista ed un certo numero di direzioni precedenti. I pesi assegnati ad ogni direzione variano dinamicamente in funzione della velocità: se l'utente è fermo viene considerata la sola direzione di vista, mentre durante il movimento viene assegnato a $\vec{V}$ un peso inversamente proporzionale alla velocità. Così facendo si ottiene un buon controllo sulla direzione a basse velocità, evitando al contempo sbalzi repentini di direzione nel caso in cui l'utente distolga lo sguardo dalla destinazione mentre si sposta ad alte velocità.

Degna di nota è anche la tecnica proposta in [39], dove l'utente naviga nella scena alla guida di un monopattino elettrico virtuale. Il monopattino è fisicamente costituito da una Wii Balance Board e da un touchscreen. La Balance Board viene impiegata per il controllo del veicolo, mentre sullo schermo viene visualizzata un mappa interattiva, utile sia per il "*wayfinding*", sia per realizzare spostamenti veloci su lunghe distanze (con modalità simili al teletrasporto). L'impiego di mappe interattive per facilitare l'orientamento nella scena e fornire modalità di navigazione alternative è abbastanza comune. Approfondiremo questo argomento nel paragrafo 1.5.6.

[8] Il vettore $\vec{W}$ deve essere ortogonale a $\vec{V}$ e parallelo al terreno

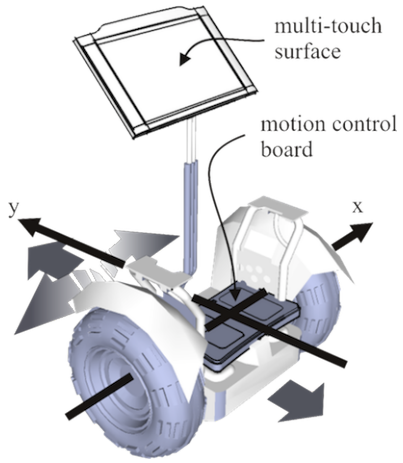


Figura 1.5: La tecnica "Human-Transporter"

1.5.3 Speed-coupled flying

Variante del *Flying vehicle*, questa metafora consente sia una navigazione immersiva a bassa velocità, sia di percorrere lunghe tratte a velocità sostenuta, fornendo al contempo una visione d'insieme (dall'alto) della scena stessa [37]. L'idea è quella di mettere in relazione altezza e inclinazione della camera con la velocità di navigazione: a basse velocità la camera viene mantenuta orizzontale e ad "altezza d'uomo" mentre, man mano che la velocità aumenta, la camera tende ad alzarsi ed inclinarsi verso il terreno, in modo da fornire una visione d'insieme della scena. Quando l'utente rallenta o non impartisce più comandi avviene l'inverso: la camera gradualmente ripristina i valori iniziali di altezza e inclinazione, con un effetto simile ad una planata. Per le sue caratteristiche, questa metafora è particolarmente indicata in ambienti di grandi dimensioni, mentre è poco adatta a scene piccole o che modellano ambienti chiusi.

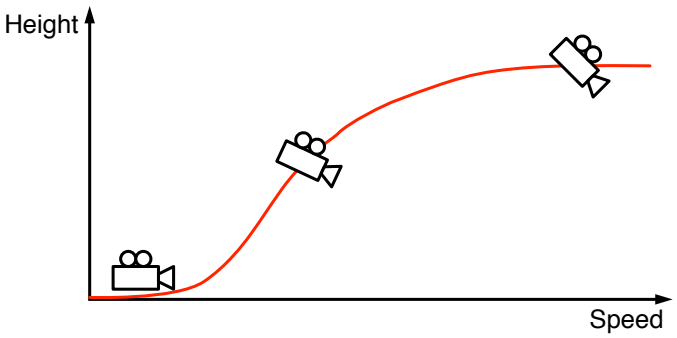


Figura 1.6: Speed-coupled flying

Tecniche di interazione

In [37] viene proposto il mouse come input device, ma la stessa tecnica lì descritta può essere usata anche con altri dispositivi a due gradi di libertà, quali i touchscreen o le

tavolette grafiche. In alternativa si possono adattare le tecniche di interazione del *Flying vehicle*.

1.5.4 Orbiting

Dato un oggetto (o un punto) di interesse all'interno della scena, consideriamo una sfera invisibile centrata in esso. In questa metafora la camera è posizionata sulla superficie della sfera e può muoversi esclusivamente su tale superficie. La camera è inoltre sempre orientata verso l'oggetto di interesse. Eventuali zoom (in/out) possono essere quindi pensati come variazioni del raggio della sfera. Questa metafora è quindi adatta a tutte quelle situazioni in cui l'attività dell'utente consiste nel visionare un oggetto di interesse da varie angolazioni.

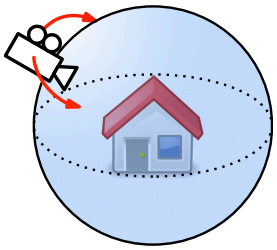


Figura 1.7: Orbiting

Un problema si presenta quando nelle vicinanze dell'oggetto di interesse sono presenti altri elementi della scena. In tali circostanze può infatti capitare che uno o più di tali elementi si interpongano tra la camera e l'oggetto di interesse, occludendolo. Una possibile soluzione consiste nel non visualizzare gli elementi posti nelle vicinanze dell'oggetto considerato, oppure creare una speciale modalità di orbiting in cui viene visualizzato il solo oggetto di interesse.

Tecniche di interazione

In ambito desktop, ed in particolare utilizzando dispositivi di puntamento a due gradi di libertà (mouse, tavolette grafiche, touchscreen ecc.), l'orbiting viene generalmente controllato mediante *dragging*. Lo zoom viene tipicamente attivato "draggando" con un tasto differente, sfruttando la rotella del mouse o, nel caso di touchscreen con tecnologia multitouch, usando una gesture di tipo *pinch-zoom*.

Per quanto riguarda l'utilizzo di sistemi di tracking, è comune usare una gesture di *grab*: l'utente afferra virtualmente l'oggetto e trascina la mano come per ruotarlo. In alternativa, sfruttando un head mounted display ed il solo tracking della testa, si può pensare di controllare la posizione della camera sulla sfera in funzione dell'orientamento della testa,

come proposto in [21]. Ad esempio: inclinando la testa verso l'alto viene visualizzata la parte inferiore del modello, mentre inclinando la testa verso il basso viene visualizzata la parte superiore, e così via... Viene però fatto notare che, poiché i movimenti della camera non corrispondono a quelli effettuati dal corpo, alcuni utenti possono accusare un senso di nausea.

1.5.5 Eyeball in hand e Scene in hand

"**Eyeball in hand**" e "**Scene in hand**" sono state tra le prime metafore di navigazione proposte in letteratura [40]. In entrambe la scena tridimensionale è "compresente" con la stanza reale, ed entrambe richiedono l'utilizzo di un dispositivo a sei gradi di libertà impugnato dall'utente. Le due metafore si differenziano sul significato attribuito a tale dispositivo.

Nella metafora "*Eyeball in hand*" il dispositivo impugnato dall'utente rappresenta un "occhio" attraverso il quale si può vedere la scena (altrimenti invisibile). L'immagine della scena rilevata dall'occhio viene visualizzata a schermo. Poiché c'è una corrispondenza biunivoca tra i movimenti del dispositivo e quelli dell'*eyeball*, la navigazione nell'ambiente virtuale avviene spostandosi all'interno della stanza reale. Questa è anche la principale limitazione di questa metafora: le dimensioni della scena sono limitate sia dal raggio di azione del dispositivo (che quindi deve essere preferibilmente wireless) sia dalla dimensione dell'ambiente di lavoro. In pratica l'intera scena non può avere una superficie superiore a qualche metro quadrato. Altra debolezza deriva dal fatto che l'utente, in alcune occasioni, può essere costretto ad assumere posizioni scomode per ottenere il punto di vista desiderato.

Spostare la camera all'interno della scena è equivalente allo spostare la scena mantenendo fissa la camera: è questa l'idea alla base della metafora "*Scene in hand*". A differenza di "*Eyeball in hand*", il dispositivo impugnato dall'utente rappresenta la scena stessa: traslazioni e rotazioni del dispositivo vengono così tradotte in traslazioni e rotazioni della scena. Si noti però che, per uno stesso movimento del dispositivo di input, la scena subisce una trasformazione inversa a quella che subirebbe con "Eyeball in hand": ad esempio, in un'ipotetica scena costituita da una stanza, una rotazione verso l'altro farebbe visualizzare il pavimento anziché il soffitto. Al fine di migliorare l'usabilità della metafora, viene proposto l'utilizzo di una sorta di frizione virtuale attivata tramite un pulsante: le trasformazioni vengono applicate alla scena solo se il pulsante è premuto, permettendo così di rilasciare il pulsante, riportare il dispositivo nella sua posizione di riposo ed eseguire di nuovo l'azione. Questa metafora risulta comunque inadatta a navigazioni arbitrarie all'interno di scene complesse, mentre fornisce buoni risultati in scene costituite da un

singolo elemento di interesse, dove l'attività principale dell'utente consiste nel visionare tale modello da varie angolazioni.

1.5.6 Map-based navigation

Fino ad ora abbiamo esaminato un solo aspetto della navigazione, quello riguardante lo spostamento all'interno dell'ambiente virtuale. Esiste però anche un aspetto cognitivo, detto *wayfinding*, ovvero il processo mentale attraverso il quale l'utente determina il percorso per raggiungere una certa destinazione. Alcune metafore, come il *Flying vehicle* o lo *Speed-coupled flying*, agevolano per loro stessa natura questo processo, fornendo una vista dall'alto della scena. Non tutte le metafore hanno però questa peculiarità (si pensi al *Walking*), oppure questa può non essere sufficiente. È allora necessario aiutare l'utente per mezzo di indicazioni aggiuntive, ad esempio sotto forma di mappe virtuali. Queste mappe possono essere statiche (si limitano a rappresentare una versione semplificata ed in scala ridotta della scena, e la posizione dell'utente all'interno di essa) oppure interattive. In quest'ultimo caso la mappa diventa l'elemento chiave per una nuova categoria di metafore di navigazione. In base alle modalità con cui l'utente interagisce con la mappa, queste metafore possono essere classificate in tre grandi categorie:

- **Camera-based navigation** [1]: posizione e orientamento della camera vengono rappresentati sulla mappa per mezzo di un avatar manipolabile dall'utente. Ogni trasformazione applicata all'avatar corrisponde ad una trasformazione della camera. L'applicazione di queste trasformazioni può avvenire immediatamente[9] (cioè in concomitanza con la manipolazione dell'avatar) oppure si può aspettare il termine della manipolazione e transire al nuovo stato con un'animazione. Si può anche dare all'utente la possibilità di ruotare o traslare la mappa, purché gli avatar in essa presenti mantengano la stessa posizione e lo stesso orientamento rispetto alla mappa stessa. Una delle più note metafore appartenenti a questa categoria è il "**Word in miniature**" [35, 29].
  Questo tipo di metafore sono estremamente versatili. Sono ad esempio applicabili in contesti con camere multiple: basta inserire nella mappa un avatar per ciascuna camera in uso; manipolando uno di questi avatar, verrà attivata la camera corrispondente. Esistono anche implementazioni che mettono a disposizione dell'utente mappe multiple, al fine di rappresentare regioni molto distanti tra loro oppure lo stato della scena in differenti istanti temporali [35]: spostando l'avatar da una mappa all'altra si ottiene l'effetto di un teletrasporto nello spazio o nel tempo.

[9] Soluzione che può risultare fastidiosa per gli utenti.

- **Scene-based navigation**[10] [1]: anziché manipolare l'avatar corrispondente alla camera si manipola la mappa stessa. L'avatar è quindi immutabile e posto al centro della mappa. Si noti che, per eseguire una certa trasformazione della camera, occorre applicare alla mappa la trasformazione inversa. Ad esempio, per spostare la camera in una certa direzione, con l'approccio *camera-based* basta trascinare l'avatar nella direzione desiderata, mentre nella *scene-based navigation* occorre trascinare la mappa nella direzione inversa.

- **Point of Interest based navigation**: nella mappa è presente un insieme di punti di interesse. Selezionando uno di questi punti la camera viene spostata automaticamente in prossimità dello stesso. Questa transizione avviene in maniera completamente automatica con modalità simili ad un "teletrasporto" oppure seguendo un percorso prestabilito. Questo approccio può essere usato indipendentemente o fungere da complemento ai precedenti. La discussione sulla *Point of Interest based navigation* verrà approfondita nel paragrafo 1.5.7 a pagina 19.

Si noti infine che gli approcci *camera-based* e *scene-based* si basano sugli stessi principi di "*Eyeball in hand*" e "*Scene in hand*", applicati però ad una mappa interattiva.

Figura 1.8: Esempio di camera-based e scene-based navigation

Tecniche di interazione

In ambito desktop, l'interazione con la mappa avviene tipicamente tramite mouse o un altro dispositivo di puntamento a due gradi di libertà: gli elementi della mappa vengono selezionati con un "click" e manipolati tramite "drag".

[10] Chiamata "View-based" in [1]

Più interessanti sono le soluzioni adottate in ambienti immersivi. In particolare esamineremo alcuni dei problemi riscontrati nello sviluppo del "**Word in miniature**" (WIM) [35, 29] e le soluzioni adottate, in quanto tali problemi ricorrono spesso nello sviluppo di applicazioni per ambienti immersivi e le soluzioni proposte hanno valenza generale. Il WIM utilizza una mappa tridimensionale che, grazie alla visione stereoscopica e al tracking delle mani, può essere manipolata direttamente dall'utente. Nelle prime implementazioni la mappa veniva visualizzata e gestita come un generico elemento fluttuante della scena. Il primo problema che si è presentato consisteva nel fatto che, nonostante la visione stereoscopica, la manipolazione del WIM risultava difficoltosa a causa dell'assenza di feedback tattile. Tale problema può essere in parte risolto legando la posizione della mappa alla mano non dominante. Così facendo l'utente riesce a percepire con maggiore accuratezza la posizione del WIM nello spazio, rendendone più agevole la manipolazione. Ciò si ottiene grazie alla propriocezione, cioè la capacità di percepire e riconoscere la posizione del proprio corpo nello spazio anche senza il supporto della vista. In alternativa, si può usare un oggetto fisico impugnabile dall'utente (ad esempio un ferma-blocco su cui sono stati installati dei sensori di tracking) come supporto per il WIM, dando così la possibilità di spostare e ruotare liberamente la mappa.

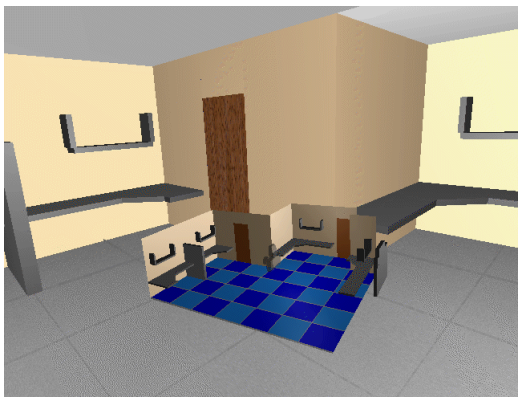


Figura 1.9: Word in miniature

In assenza di hand-tracking si può affiancare al dispositivo di visualizzazione principale un touchscreen, con cui visualizzare e manipolare la mappa.

1.5.7 Navigazione basata su punti di interesse

Molte applicazioni danno la possibilità all'utente di selezionare un "**punto di interesse**" (*Point Of Interest - POI*) all'interno della scena, per poi spostarsi automaticamente in prossimità di esso. L'esempio classico è quello del museo virtuale: ogni opera esposta rappresenta un punto di interesse e la navigazione all'interno del museo può avvenire in maniera completamente automatica, muovendo la camera da un punto di interesse

all'altro per mezzo di una animazione. I POI possono essere prestabiliti e immutabili oppure definibili dall'utente. Le metafore appartenenti a questa categoria si differenziano in base a come viene scelto il POI e come vengono determinati posizione e orientamento della camera al termine dello spostamento.

In genere il POI viene scelto semplicemente selezionando un elemento della scena. In particolare la metafora di selezione più adatta a tale scopo è il "**ray-casting**" (si veda il paragrafo 1.6.2 a pagina 24), perché consente di individuare uno punto specifico dell'elemento selezionato da utilizzare come POI. Quando viene usato un mouse come dispositivo di input questa metafora è detta "**Click and fly**". Un'alternativa comune (in particolare in ambito desktop) prevede che l'utente selezioni una regione della scena anziché uno specifico elemento (ad esempio disegnando un rettangolo sullo schermo). In questo caso la camera dovrà avvicinarsi il più possibile all'area selezionata, cercando al contempo di mantenere un'inquadratura completa della stessa. Si parla quindi di "**Area of interest**" e la metafora corrispondente è detta "**Region zoom**".

In alternativa i punti di interesse possono essere prestabiliti e selezionabili dall'utente per mezzo di un componente grafico di supporto, quale una mappa o una semplice lista.

Per quanto riguarda la posizione finale della camera, essa può essere ricavata a partire dai seguenti parametri:

1. Il punto di interesse $\dot{P}$.
2. La distanza finale δ tra la camera e $\dot{P}$.
3. L'orientamento finale della camera, dato da un versore $\vec{D}$.

La posizione $\dot{C}$ della camera al termine dello spostamento è data da:

$$\dot{C} = \dot{P} - (\delta \cdot \vec{D})$$

I parametri δ e $\vec{D}$ possono essere prestabiliti o regolabili (in un qualche modo) dall'utente. È comune scegliere come direzione $\vec{D}$ il vettore inverso alla normale alla superficie nel punto $\dot{P}$.

Per la metafora "*Region zoom*" si procede come segue: $\vec{D}$ corrisponde alla direzione della retta che, partendo dal *viewpoint*, interseca il *near plane* in corrispondenza del centro della regione selezionata. Il punto $\dot{P}$ giace su tale retta, ad una distanza σ dalla camera. Esistono vari metodi per la scelta di σ [17]: il più semplice consiste nell'individuare l'intersezione tra la retta e gli elementi della scena (punto $\dot{P}_3$ in figura 1.10). Tale punto potrebbe però non esistere (si pensi ad un arco), oppure oltrepassare la regione desiderata. L'approccio più conservativo consiste invece nell'adottare come σ la minima tra le profondità dei frammenti della regione (punto $\dot{P}_1$ in figura 1.10). In alternativa, σ può essere

calcolato come il valor medio dei valori di profondità dei frammenti, oppure il valore più frequente (punto $\dot{P}_2$ in figura 1.10). A questo punto si può ricavare $\dot{C}$ come descritto in precedenza.

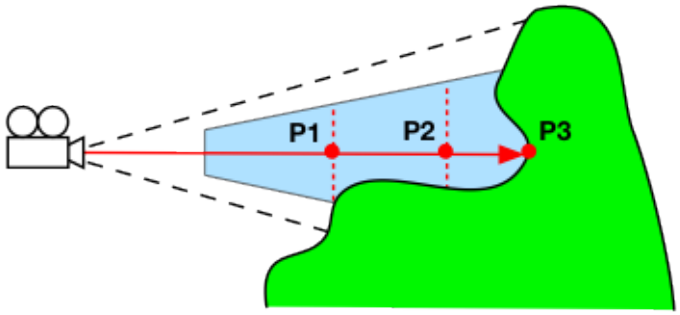


Figura 1.10: Determinazione di un punto di interesse nella metafora "*Region zoom*"

Tecniche di interazione

Poiché il metodo più comune per selezionare un punto di interesse è il *ray-casting*, valgono le stesse tecniche sviluppate per tale metafora (paragrafo 1.6.2 a pagina 24).

Oltre a queste, una delle migliori tecniche studiate appositamente per la navigazione basata su aree di interesse è "**Navidget**" [17, 18, 20]. L'idea chiave di questa tecnica consiste nell'impiego di un controllo grafico tridimensionale per selezionare l'area di interesse e controllare l'orientamento finale della camera (figura 1.11). Il widget è immerso nella scena e richiamabile attraverso la pressione di uno specifico tasto oppure mantenendo premuto un tasto per un certo periodo di tempo. Una volta attivato, esso consiste in una sfera attorno cui orbita l'avatar della camera. La sfera rappresenta la porzione di l'area di interesse visibile al termine dello spostamento, e può essere ridimensionata per mezzo di apposite "linguette". Spostando l'avatar sulla sfera si determina la direzione finale di vista. È anche possibile attivare una finestra di preview per l'ispezione a distanza.

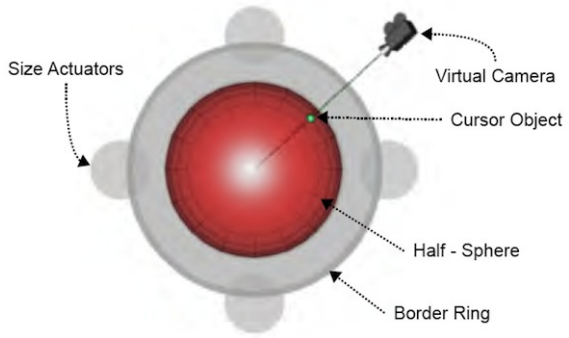


Figura 1.11: Navidget

Navidget è utilizzabile con qualsiasi dispositivo di puntamento a due o più gradi di libertà (mouse, touchscreen, "air-mouse" ecc...), ed è impiegabile in ambienti desktop, mobile, collaborativi ed immersivi.

Figura 1.12: Navidget in ambienti collaborativi

1.6 Metafore di selezione

Gran parte delle metafore per selezione degli elementi della scena possono essere classificate in due grandi famiglie, chiamate "**Virtual hand**" e "**Virtual pointer**".

1.6.1 Virtual hand

Viene inserita nella scena una rappresentazione virtuale della mano dell'utente, sotto forma di un cursore di qualche tipo. La selezione di un oggetto avviene muovendo il cursore in modo da farlo intersecare con l'oggetto scelto e confermando in un qualche modo la selezione (es. premendo un pulsante).

Tecniche di interazione

Questa metafora è usata quasi esclusivamente in sistemi immersivi e in presenza di hand tracking o interfacce aptiche. Le varie tecniche si distinguono principalmente in base alla funzione che associa posizione e orientamento della mano reale alla posizione e all'orientamento della mano virtuale [32]. Ne caso più semplice, tale funzione sarà l'identità: ad ogni movimento della mano corrisponderà un movimento di pari entità del cursore. La principale limitazione di questa tecnica consiste nell'impossibilità di selezionare oggetti al di fuori del raggio di azione delle braccia dell'utente, costringendolo così a spostarsi nella scena.

La soluzione proposta in [33], chiamata "**Go-Go**", utilizza una funzione a tratti: per estensioni del braccio inferiori ad una certa soglia (tipicamente 2/3 della lunghezza complessiva) la funzione ha lo stesso andamento dell'identità, mentre per estensioni superiori si impiega una funzione quadratica, in modo da raggiungere oggetti distanti. Si noti comunque che il raggio d'azione resta comunque limitato. C'è da dire che esistono varianti che consentono di estendere la mano virtuale ad una distanza arbitraria, tuttavia maggiore sarà l'estensione, maggiore sarà la difficoltà di controllo.

Il problema della selezione di oggetti distanti può essere attenuato impiegando sistemi di visione stereoscopica e dispositivi di input con feedback aptico. Un ulteriore miglioramento si ottiene con l'utilizzo di guide aptico/visive, come proposto in [38]. L'idea è quella di immaginare ogni oggetto selezionabile come circondato da due regioni sferiche. Quando il cursore entra nella regione sferica più esterna, viene attivato un feedback visivo, per informare l'utente della prossimità all'oggetto. Avvicinando ulteriormente il cursore si entra nella regione sferica interna, nella quale viene attivata la "guida aptica": viene cioè generata una forza che attrae il cursore verso l'oggetto, con un'intensità inversamente proporzionale alla distanza dall'oggetto stesso (e quindi sempre più intensa man mano che il cursore si avvicina). Ovviamente i due feedback vengono disattivati una volta avvenuta la selezione o se l'utente decide di spostare la mano virtuale al di fuori delle due regioni. Il raggio della regione sferica interna (quella che attiva la guida aptica) deve essere scelto in modo da evitare la sovrapposizione con le regioni degli oggetti vicini. In caso contrario le forze attrattive si sommerebbero, inducendo l'utente in confusione. Segue che l'utilizzo di questa tecnica risulta problematico in scene con molti oggetti selezionabili posti vicini tra loro.

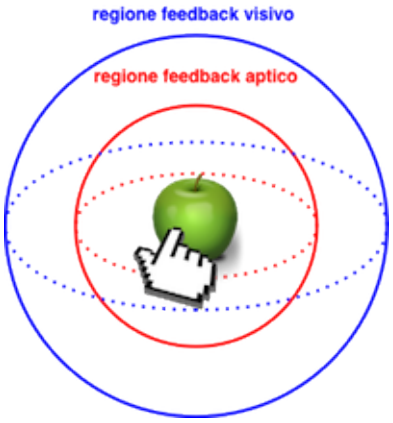


Figura 1.13: Selezione tramite Virtual hand con guide aptico-visive

1.6.2 Virtual pointer

L'altra importante categoria di metafore di selezione è il "**Virtual pointer**", nella quale l'utente seleziona un oggetto semplicemente puntandolo. La più nota di queste metafore è il "**Ray-casting**", che prevede la proiezione di una semiretta, detta *raggio*, nella scena. Il raggio viene proiettato a partire da un cursore controllato dall'utente. La selezione avviene orientando il raggio in modo da farlo intersecare con l'oggetto scelto e confermando in un qualche modo tale scelta (ad esempio premendo un pulsante). Se il raggio interseca più di un oggetto, viene scelto quello più vicino al *viewpoint*. Il *Ray-casting* è lo "standard de facto" per la selezione in ambito desktop, grazie alla sua semplicità e precisione. In ambienti immersivi questa precisione viene meno quando si cerca di selezionare oggetti particolarmente piccoli e/o distanti. Questo problema deriva sia dall'imprecisione e dal rumore dei sistemi di tracking, sia dall'impossibilità per gli utenti di restare perfettamente immobili.

Sono state proposte diverse varianti del *Ray-casting* per porre rimedio a questo problema. Una di queste, chiama "**Sticky-ray**", prevede che un oggetto possa essere selezionato se si trova in prossimità del raggio, anche in assenza di intersezione [34]. L'idea è quella di individuare l'oggetto selezionabile più vicino al raggio, chiamato oggetto attivo, e di curvare il raggio verso tale oggetto, come se venisse attratto da una forza magnetica o gravitazionale generata dall'oggetto attivo. Tra tutti gli oggetti selezionabili, l'oggetto attivo è quello con la minima distanza dal raggio. Nel caso in cui più oggetti abbiano la stessa distanza dal raggio, viene scelto quello più vicino al *viewpoint*. Il nome di questa metafora deriva dal fatto che il raggio resta "attaccato" all'oggetto attivo fintanto che non viene individuato un oggetto più vicino o non viene superata una certa distanza massima.

Figura 1.14: Sticky-ray

Un'altra possibilità consiste nel sostituire il raggio con un volume di selezione[11]. La più nota metafora di questo tipo è il "**Cone casting**", nel quale il raggio viene sostituito da un cono di lunghezza infinita. Si noti che con l'aumentare della distanza dal *viewpoint* aumenta anche la dimensione del cono, e con essa la probabilità che più oggetti si trovino all'interno del volume di selezione. Occorre quindi definire una politica discriminatoria: in caso di conflitto viene selezionato l'oggetto più vicino all'asse del cono e, a parità di distanza, quello più vicino al *viewpoint*.

Tecniche di interazione

Le tecniche di interazione per la metafora "*Virtual pointer*" si distinguono in base a come vengono definiti origine e direzione del volume di selezione.

Partiamo dalle tecniche relative al "*Ray-casting*". In ambito desktop, il raggio deve essere generato in corrispondenza del cursore del mouse. Per ottenere ciò basta utilizzare la posizione della camera come origine del raggio, mentre la direzione è data dal vettore che congiunge tale origine con il punto ottenuto proiettando la posizione del cursore sul *near plane*. Per quanto riguarda gli ambienti immersivi possono essere prese in considerazione diverse possibilità, in base al sistema di tracking disponibile:

- *Head tracking*: l'origine del raggio corrisponde al *viewpoint*, mentre la direzione si ricava dall'orientamento della testa.
- *Hand tracking*: come nelle metafore "*Virtual hand*" viene introdotta nella scena una rappresentazione virtuale della mano dell'utente, da cui ha origine il raggio. La direzione può essere ricavata dall'orientamento della mano o del braccio.

Avendo a disposizione entrambi, si può prendere in considerazione la tecnica proposta in [30], che prevede l'utilizzo di gesture come metodo di selezione semplice e familiare

[11] Di fatto, anche "Sticky-ray" può essere considerato appartenente a questa categoria: è infatti equivalente ad una metafora con volume di selezione cilindrico.

per l'utente. Come sarà presto possibile rendersi conto, l'utente tende spontaneamente ad eseguire queste gesture tenendo aperto un solo occhio, che chiameremo *dominante*. Infatti questa tecnica sfrutta gli effetti ottici che si ottengono proiettando una scena tridimensionale su un piano bidimensionale. Per tale ragione è stata chiamata "**Image plane based selection**". Segue anche che non sono necessari dispositivi di visualizzazione stereoscopica. Dal punto di vista implementativo si fa ancora uso *ray-casting*, in quanto per ogni gesture si può ricavare un raggio con il quale eseguire il test di selezione.
La prima gesture proposta in [30], detta "*Sticky Finger*" (figura 1.15 *(a)*), ricalca il concetto di "*Virtual Pointer*": l'utente usa il dito indice per puntare l'oggetto da selezionare. La selezione vera e propria avviene proiettando nella scena un raggio corrispondente al vettore che congiunge l'occhio dominante dell'utente[12] con la punta del suo indice. Si noti che la stessa tecnica era già stata descritta in relazione al Flying-vehicle (pagina 10), ed è nota anche come "*Crosshair*".
La seconda gesture, detta "*Head Crusher*", prevede che l'utente posizioni pollice e indice attorno dell'oggetto desiderato, come mostrato in figura 1.15 *(b)*. Stavolta il raggio, sempre partendo dall'occhio dominante, passa attraverso il punto centrale tra pollice e indice.
Nel "*Framing Hands*" l'utente usa entrambe le mani per formare una sorta di cornice attorno all'oggetto da selezionare (figura 1.15 *(c)*). Il raggio è definito dalla posizione dell'occhio dominante e dal centro della cornice. Si noti che questa gesture può essere impiegata anche per la selezione di un gruppo di oggetti vicini.
Infine nel "*Lifting Palm*" l'utente stende il braccio e rivolge il palmo della mano verso alto, in modo da avere l'illusione che l'oggetto si trovi sopra il suo palmo. I punti dai quali ricavare il raggio sono la posizione dell'occhio dominante e la posizione del palmo della mano; a quest'ultimo punto deve essere però aggiunto un piccolo offset verso l'alto.

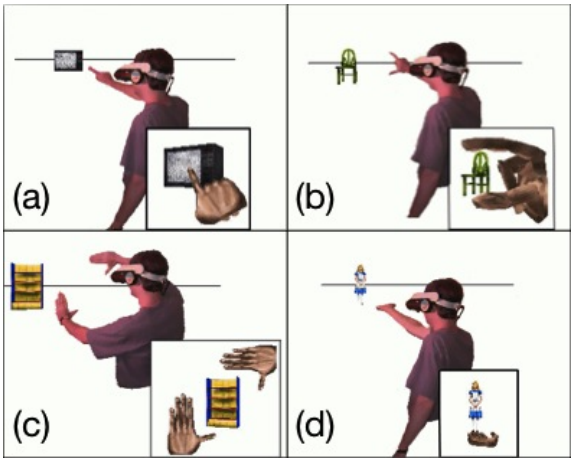


Figura 1.15: Image plane based selection

La tecnica "*Crosshair*" può essere applicata anche al cone-casting. In questo contesto

[12] La cui posizione può essere approssimata a partire dalla posizione della testa.

è nota come "**Aperture selection**" [13]. In accordo a quanto detto in precedenza, la direzione del cono è data dal vettore che congiunge l'occhio dominante dell'utente con la posizione di un oggetto di puntamento da lui impugnato (può essere anche un dito della mano). La dimensione del cono è invece proporzionale alla lunghezza di tale vettore, ovvero alla distanza tra l'occhio e il puntatore. Anziché visualizzare il cono, [13] suggerisce di disegnare nella scena un mirino circolare in corrispondenza del puntatore. Il diametro del mirino deve essere pari al diametro della sezione circolare del cono a quella data distanza dal vertice.

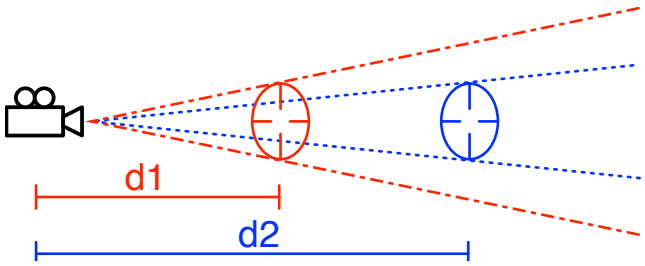


Figura 1.16: Aperture selection

1.7 Metafore di manipolazione

Molte delle metafore per la manipolazione di oggetti che vedremo sono già state descritte nei paragrafi precedenti. Infatti, la selezione può essere considerata come la prima fase di un'operazione di manipolazione, ed è quindi comune trovare metafore che si applicano ad entrambe, se non addirittura anche alla navigazione. Inizieremo la trattazione dalle metafore già note, descrivendo le sole caratteristiche relative alla manipolazione, per poi passare a quelle ideate appositamente per questo tipo di interazione.

1.7.1 Ray-casting, Virtual hand e HOMER

Il *Ray-casting* può essere impiegato anche nella manipolazione degli elementi della scena [4]: una volta selezionato un oggetto, questo resta legato al raggio seguendone gli spostamenti. Questo approccio ha però molte limitazioni: ad esempio non esiste un modo semplice e immediato per ruotare l'oggetto attorno ad un asse diverso da quello costituito dal raggio stesso, né per avvicinare o allontanare l'oggetto rispetto all'osservatore (a meno di non ricorrere all'uso di appositi pulsanti). Al contrario, tutte le metafore basate su *Virtual hand* offrono buoni risultati anche nel campo della manipolazione (in ambienti immersivi), grazie alla semplicità con cui è possibile convertire posizione e orientamento della mano in trasformazioni da applicare all'oggetto.

Si può allora pensare ad una metafora che combini la selezione tramite *Ray-casting* con la manipolazione attraverso la *Virtual hand*, sfruttando così i vantaggi di entrambi. Questa è

l'idea alla base della metafora chiamata "**HOMER**" (*Hand-centered Object Manipulation Extending Ray-casting*) [4]: una volta selezionato un oggetto tramite ray-casting, il raggio scompare e viene sostituito da una mano virtuale che si "lega" all'oggetto nel punto di intersezione con il raggio. Così facendo è possibile manipolare un elemento a distanza con la stessa semplicità e precisione di una manipolazione locale, anche se limitatamente al raggio di azione del braccio dell'utente. Per ovviare a questa limitazione viene proposto di utilizzare dei pulsanti per modificare la distanza tra utente e oggetto manipolato.

1.7.2 Worlds In Miniature

Abbiamo già parlato del Worlds In Miniature come metafora di navigazione. La navigazione all'interno di un WIM può essere vista come un caso particolare di manipolazione di oggetti, nel quale l'oggetto manipolato è l'avatar dell'utente. La stessa procedura può quindi essere applicata ad ogni altro elemento della scena: per modificare la posizione o l'orientamento di un oggetto, l'utente "afferra" il corrispondente proxy nella miniatura e lo sposta o lo ruota come desiderato. L'oggetto a scala reale nella scena subirà quindi le stesse trasformazioni applicate al proxy nella miniatura. Questa soluzione si rivela molto efficace per manipolazioni a grana grossa, ovvero su oggetti relativamente grandi e spostamenti consistenti, ma presenta serie limitazioni quando si devono manipolare oggetti di piccole dimensioni, oppure quando è necessario agire con precisione. Questi problemi possono essere risolti in vari modi, tra cui:

- Consentendo all'utente di regolare la scala della miniatura [26].
- Imponendo dei vincoli sulle trasformazioni, ad esempio impiegando tecniche di collision detection per evitare compenetrazioni, oppure introducendo una griglia virtuale che vincola la posizione degli oggetti alla griglia stessa ("*snap-to-grid*").

1.7.3 Object in hand

Fino a questo momento abbiamo considerato come possibili manipolazioni le sole traslazioni e rotazioni. Questo tipo di trasformazioni non modificano l'oggetto in sé, bensì il suo stato in relazione al contesto in cui è inserito. Le altre (es. colorazione, deformazioni ecc.), al contrario, sono indipendenti dal contesto in cui è inserito l'oggetto. Per gli esseri umani è naturale effettuare questo tipo di manipolazioni sfruttando entrambe le mani: l'oggetto viene afferrato con la mano non dominante (togliendolo quindi dal suo contesto originario) e manipolato con la mano dominante. In altri termini la mano dominante lavora rispetto al sistema di riferimento definito dalla mano non dominante, sfruttando la *propriocezione*. Queste osservazioni sono alla base della metafora nota come "**Object in**

Hand" [8], la cui implementazione tipica richiede il tracking della mano non dominante e di un'interfaccia aptica (ad esempio un PHANToM) controllata dalla mano dominante e rappresentata nella scena sotto forma di cursore. L'interazione avviene come segue: una volta selezionato un oggetto, l'utente lo rimuove dal suo contesto stringendo a pugno le dita della mano non dominante (come se lo stesse effettivamente impugnando). L'oggetto viene quindi portato in primo piano e "legato" alla mano non dominante, seguendone movimenti e rotazioni. L'utente può quindi avvicinare l'oggetto al cursore ed effettuare la manipolazione con l'aiuto dell'interfaccia aptica.

1.7.4 Voodoo Dolls

"**Voodoo Dolls**" è una metafora per la manipolazione a distanza di oggetti in un ambiente virtuale immersivo [31]. L'utente interagisce utilizzando entrambe le mani, sfruttando il principio secondo cui la mano dominante (che d'ora in poi supporremo essere la destra) lavora in un sistema di coordinate definito da quella non dominante. L'interazione avviene come segue: una volta selezionato un oggetto per mezzo di una delle tecniche *Image plane based* (descritte a pagina 25), ne viene creata una copia in scala ridotta, detta *doll*. Questa resta legata alla mano, seguendone movimenti e rotazioni, e può essere scambiata da una mano all'altra. Tutte le trasformazioni dipendenti dal contesto richiedono la creazione di due *doll*, una per all'oggetto da modificare (legata alla mano destra) e l'altra per il suo contesto (legata alla mano sinistra). La *doll* tenuta nella mano sinistra definisce un sistema di coordinate in base al quale vengono valutate le trasformazioni applicate alla *doll* della mano destra. In altre parole, all'oggetto selezionato vengono applicate le stesse trasformazioni effettuate sulla sua *doll*, ma non rispetto ad un sistema di riferimento assoluto, bensì rispetto a quello dell'oggetto la cui *doll* è impugnata dalla mano sinistra. Ad esempio, volendo spostare una penna su una scrivania, dobbiamo creare una *doll* della scrivania (il contesto) ed impugnarla con la mano sinistra, creare una *doll* della penna con la mano destra, ed infine poggiare la *doll* della penna sulla *doll* della scrivania.

Figura 1.17: Voodoo dolls

1.7.5 3D Graphical User Interface

Molti autori trattano le interfacce grafiche 3D come un caso particolare della manipolazione di oggetti. Questo perché i componenti grafici vengono implementati e gestiti in maniera simile agli altri oggetti della scena. Il principale problema delle interfacce grafiche tridimensionali deriva dalla difficoltà degli utenti nell'interagire con oggetti "fluttuanti", specie in assenza di visione stereoscopica e feedback aptico. Questo problema può essere risolto tramite la *propriocezione*. In particolare, tra le metafore che sfruttano questo principio, *Object in Hand* si è dimostrata particolarmente efficace: i menu e le finestre di dialogo vengono "legati" alla mano non dominante, mentre l'interazione vera e propria avviene per mezzo del PHANToM, potendo così contare anche sul feedback aptico. In assenza di un'interfaccia aptica, l'attivazione dei *widget* può avvenire tramite ray-casting [26].

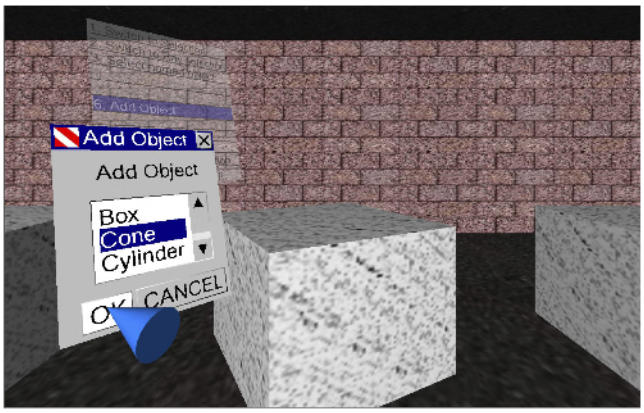


Figura 1.18: Interfaccia grafica 3D controllata dalla metafora "Object in Hand"

Parte II

Il framework di interazione

Capitolo 2

Introduzione a XVR ed al framework di interazione

Il progetto di tesi consiste nella progettazione e realizzazione di un framework per semplificare l'implementazione dell'interazione utente nelle applicazioni XVR, supportando al contempo dispositivi di input di natura diversa. Nel framework sono centrali i concetti di *metafora* e *tecnica di interazione*, introdotti nel capitolo 1. Esso tuttavia non si limita a fornire una libreria di metafore e tecniche di interazione, bensì introduce un livello di astrazione sopra XVR che incoraggia la scrittura di programmi basati sui paradigmi *Object-Oriented* e *Event-driven*. Particolare attenzione è stata posta nel rendere il framework uno strumento flessibile, capace di adattarsi alle necessità di un'ampia gamma di applicazioni.

Il framework è stato implementato in linguaggio S3D (il linguaggio di scripting di XVR), in accordo alle specifiche del progetto. L'utilizzo di questo linguaggio ha influenzato molte scelte, sia in fase di progettazione che di sviluppo. Per tale ragione, inizieremo la trattazione con una breve introduzione ad XVR ed al suo linguaggio di scripting, evidenziando le caratteristiche principali, i punti di forza e gli aspetti migliorabili.

2.1 XVR

XVR è un ambiente integrato per lo sviluppo di applicazioni di realtà virtuale sviluppato a partire dal 2001 presso il laboratorio PERCRO della Scuola Superiore Sant'Anna di Pisa [6]. XVR offre una vasta serie di funzionalità per la realizzazione di ambienti virtuali, quali un motore grafico tridimensionale, cluster rendering, supporto a vari sistemi di visualizzazione stereoscopica, suoni olofonici, e supporto ad una ampia varietà di dispositivi in input come trackers, mouse 3D ed interfacce aptiche.

Lo sviluppo di applicazioni XVR avviene attraverso un linguaggio di scripting dedicato (S3D), che semplifica l'implementazione di applicazioni di realtà virtuale grazie a costrutti dedicati e ad una ricca libreria standard. I programmi S3D vengono compilati in *bytecode* ed interpretati da un apposito *player*, che allo stato attuale è composto da due moduli principali:

- Una *virtual machine*, che si occupa dell'effettiva esecuzione del *bytecode*. Questo modulo comprende anche il cuore tecnologico di XVR, come l'engine grafico 3D, l'engine multimediale ecc.
- Un controllo ActiveX, che consente l'integrazione del player sia in applicazioni desktop tradizionali che all'interno di pagine web[1]. Questo modulo si occupa anche del controllo versione: in fase di inizializzazione viene verificata la versione della VM richiesta dall'applicazione e, se questa non è presente in locale, viene scaricata da Internet in maniera completamente automatica e trasparente all'utente. Nel caso di applicazioni *web-oriented* viene scaricato anche il *bytecode* dell'applicazione e, successivamente, i file di risorsa dell'applicazione (modelli 3D, texture ecc.).

2.1.1 Struttura e ciclo di vita di una applicazione XVR

In genere lo sviluppo di una applicazione XVR inizia dal seguente template, che ricalca la struttura di ogni programma S3D:

```
#include <Script3d.h>

// Variabili globali
var CamPos = [0.0, 5.0, 30.0]; // posizione della camera
var LightPos = [0.0, 10.0, 10.0]; // posizione del punto luce
[...]

// Funzioni definite dall'utente
function CameraMoveMouse() { [...] }
function DrawGrid(col, size) { [...] }

/* CALLBACK DI XVR */

function OnDownload() {
   // codice per il download dei file di risorse
   [...]
}
```

[1] Allo stato attuale il controllo ActiveX può essere eseguito solo in un contesto Internet Explorer, ma sono in via di sviluppo plugin per altri browser.

```
function OnInit(params) {
   // codice di inizializzazione
   [...]
}

function OnFrame() {
   CameraMoveMouse(); // gestione della camera

   // Inizio rendering
   SceneBegin();

   DrawGrid([0.5, 0.5, 0.5], 100);
   [...]

   SceneEnd();
   // Fine rendering

   // aggiornamento dello stato dell'applicazione
   [...]
}

function OnTimer() {
   // Loop ad elevata frequenza
   [...]
}

function OnEvent (eventId, param1, param2) {
   // Gestione degli eventi di basso livello
   [...]
}

function OnExit() {
   // cleanup
   [...]
}
```

Nel template possiamo distinguere sei *callback*, invocate automaticamente dalla XVR-VM:

OnDownload() - È la prima funzione ad essere invocata. In essa viene inserito il codice per il download dei file di risorse necessari all'applicazione. Questi file, indipendentemente dalla loro locazione originale (server remoto o macchina locale), vengono copiati in una cartella temporanea creata automaticamente all'inizio dell'esecuzione dello script. Tale cartella rappresenta il percorso predefinito dal quale caricare le risorse.

OnInit(userParam) - Viene invocata al termine della fase di download e contiene il codice di inizializzazione. Può ricevere un parametro (sotto forma di stringa) corrispondente al contenuto del tag *<UserParams>* nella pagina HTML che incapsula il player XVR.

OnExit() - Viene eseguita al termine dell'applicazione, tipicamente quando l'utente chiude il contenitore del controllo XVR (ad esempio la pagina Web che lo incapsula). È anche possibile (anche se insolito) richiedere esplicitamente la terminazione da programma.

Queste prime *callback* corrispondono ad istanti ben precisi del ciclo di vita dell'applicazione e vengono invocate una sola volta in tutta l'esecuzione del programma. Le due seguenti, invece, vengono eseguite periodicamente[2]:

OnFrame() - È la funzione adibita al rendering, essendo la sola a cui è permesso l'accesso al frame buffer di OpenGL. Tipicamente viene invocata con una frequenza pari a 60Hz. I comandi di rendering devono essere racchiusi tra le funzioni **SceneBegin** e **SceneEnd**. Al di fuori di tale "blocco" può essere eseguito codice generico, relativo ad esempio alla gestione della camera, all'aggiornamento dello stato dell'applicazione ecc.

OnTimer() - È caratterizzata da un'elevata frequenza di invocazione (100Hz di default, fino ad un massimo di 1KHz). Viene impiegata per compiti che richiedono un'alta frequenza di aggiornamento, come simulazione fisica, controllo di interfacce aptiche ecc.

L'ultima *callback* viene invocata a seguito della ricezione di un messaggio di Windows:

OnEvent(eventId, param1, param2) - Funzione dedicata alla gestione degli eventi, tipicamente messaggi di Windows. I parametri rispecchiano il formato di tali messaggi. È possibile generare eventi personalizzati da programma.

Sebbene questa struttura sia indubbiamente di chiara e semplice comprensione, spesso porta alla scrittura di programmi mal strutturati, nei quali le componenti logiche dell'applicazione vengono scomposte e confuse tra loro all'interno di una o più *callback*. In particolare può accadere che:

- I dati relativi al modello dell'applicazione vengano memorizzati in variabili globali e separati dal codice che implementa la "*logica di dominio*". Quest'ultimo viene

[2]La frequenze di invocazione delle due callback sono configurabili. Tuttavia tali frequenze sono da considerarsi massime e non esiste alcuna garanzia che vengano effettivamente rispettate.

in genere eseguito (nel caso peggiore direttamente, altre volte invocando funzioni appositamente definite) all'interno della *OnFrame*, ma non è raro incontrare casi in cui la *logica di dominio* viene a sua volta "sparsa" tra la *OnFrame*, la *OnTimer* e la *OnEvent*.

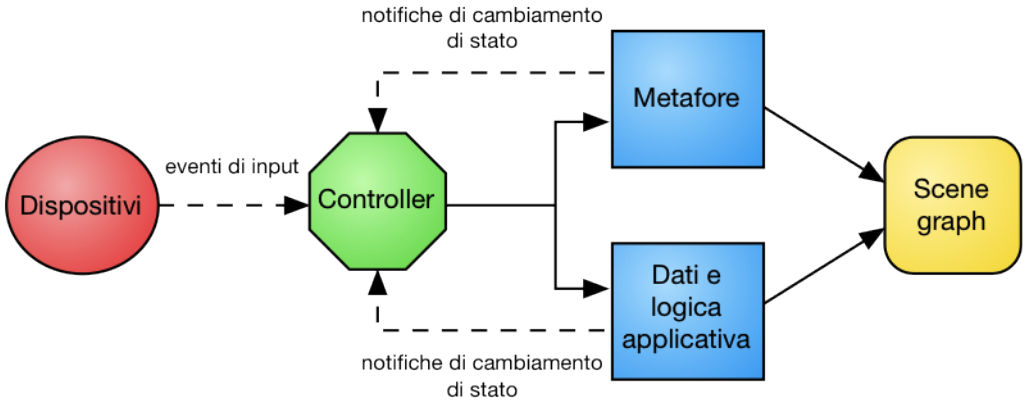
- La *OnFrame* si presenti come un unico "calderone" in cui confluisce logica applicativa, codice di rendering e gestione dell'input dell'utente, spesso difficilmente distinguibili tra loro.
- La gestione dell'input avvenga con due modalità distinte, ma talvolta coesistenti all'interno della stesso programma:
  - Nella *OnEvent*, attraverso una gestione degli eventi vera e propria. Tale gestione risulta però difficoltosa sia perché gli eventi notificati sono di basso livello (quindi occorre una vera e propria decodifica dei parametri), sia per l'estrema eterogeneità di tali eventi.
  - Nella *OnFrame*, andando a testare ad ogni frame lo stato dei dispositivi di input (*polling*).

Ovviamente non si deve pensare che non sia possibile scrivere programmi ben strutturati! Tuttavia il primo passo per riuscirci spesso consiste nella realizzazione un insieme di classi che introducono un livello di astrazione sopra XVR, in particolare per quanto riguarda la gestione dell'input. È infatti abbastanza comune che ogni programmatore definisca a tal scopo una propria libreria di classi, magari poco documentata e dalle funzionalità limitate. Questo approccio, oltre a comportare uno spreco di tempo, può compromettere la manutenibilità del software. Per tali ragioni si è deciso di includere nel framework un insieme di astrazioni e meccanismi che si propongono di semplificare lo sviluppo e di migliorare la qualità del codice prodotto.

2.2 Introduzione al framework

2.2.1 Visione d'insieme

Le classi definite nel framework modellano tre tipi fondamentali di entità: *dispositivi di input*, le *metafore di interazione* ed i *controller*.



Dispositivi di input: rappresentano delle sorgenti di eventi di input. Il loro scopo è quello di fornire un meccanismo *event-driven* per la gestione dell'input: ogni classe definisce un insieme di eventi di alto livello ed i metodi per la registrazione/rimozione dei relativi listener. Si noti che queste classi non modellano solo dispositivi fisici, ma qualsiasi tipo di sorgente di eventi di input. Ad esempio la classe *IFContainerPage*, che rappresenta il contenitore del player XVR, appartiene a questa categoria, anche se non modella un dispositivo fisico. Anche le *callback* di XVR vengono convertite in un insieme (simile ma non coincidente) di eventi, definiti ed esposti dalla classe *IFApplication*. Un simile approccio consente una gestione uniforme dell'input, indipendentemente dal fatto che questo provenga da un dispositivo che scatena eventi di sistema, da un dispositivo per cui si ha a disposizione un'API "polling-based", dai controlli presenti in una pagina Web, dalla XVR-VM o da un'applicazione remota.

Metafore di interazione: forniscono metodi di alto livello per la manipolazione dello scene graph in accordo alla semantica ed ai vincoli definiti dalla metafora. Anche le metafore possono generare eventi: in questo caso non si tratta però di eventi di input, bensì di notifiche relative al cambiamento di stato della metafora stessa.

Controller: lo scopo dei controller è quello di modificare il modello dell'applicazione in risposta ad eventi, siano essi eventi di input o notifiche di altra natura. In questo tipo di applicazioni il modello è costituito, oltre che dalle strutture dati definite dal programmatore, dalle metafore di interazione e dallo scene graph. Tutti i controller definiti nel framework implementano delle *tecniche di interazione*, quindi manipolano un oggetto metafora in risposta ad eventi di input. Viene lasciato al programmatore il compito di definire i controller che agiscono sulla parte restante del modello.

2.2.2 La classe IFApplication

La classe astratta *IFApplication* rappresenta una applicazione XVR. Essa introduce un livello di astrazione sopra XVR, in quanto:

- *Impone una metodologia di programmazione orientata agli eventi*:
  Un programma S3D non è più strutturato come un insieme di funzioni predefinite (*OnFrame*, *OnTimer* ecc.), bensì come un insieme di gestori di eventi che, agendo sul modello, realizzano le funzionalità dell'applicazione. Sebbene possa apparire più semplice implementare il corpo di una funzione predefinita rispetto al definire una classe gestore, istanziarla e registrarla, questo approccio garantisce una migliore modularità del codice.

- *Espone le callback predefinite di XVR sotto forma di eventi di alto livello*:
  Le callback di XVR sono ora eventi generati dalla classe *IFApplication*. Così facendo è possibile gestire in maniera uniforme notifiche di diversa natura, come gli eventi di input, le notifiche di cambiamento di stato di un oggetto e gli eventi relativi al ciclo di vita dell'applicazione. L'unica eccezione è costituita dalla *OnDownload* che continua ad essere una funzione, in quanto al momento della sua invocazione la classe *IFApplication* non è stata ancora istanziata.

- *Suddivide la procedura OnFrame in tre eventi distinti*:
  In un tipico programma S3D, la *OnFrame* contiene sia il codice di disegno (delimitato dalle funzioni *SceneBegi*n / *SceneEnd*), sia quello relativo all'aggiornamento dello stato dell'applicazione (posto prima e/o dopo il blocco di rendering). Esistono anche operazioni che devono necessariamente essere eseguite all'interno del blocco *SceneBegi*n / *SceneEnd*, pur non essendo relative al rendering della scena. Tali operazioni richiedono infatti che lo stato di OpenGL, ed in particolare le matrici di *model-view* e di proiezione, sia settato correttamente al momento della loro esecuzione. In presenza di queste ultime è consigliabile eseguire le restanti operazioni dopo il blocco di rendering, in modo da evitare inconsistenze al primo frame.
  Basandoci su queste considerazioni, possiamo individuare nella *OnFrame* tre fasi distinte, esposte da *IFApplication* sotto forma di eventi:

  - *Rendering*: è l'evento dedicato al rendering e, per tale ragione, viene generato all'interno del blocco *SceneBegin* / *SceneEnd*. I gestori di questo evento devono essere oggetti disegnabili, ovvero devono implementare il metodo *Draw*.
  - *PostRendering*: viene generato dopo la fase di rendering. In risposta a tale evento vengono eseguite le operazioni periodiche relative alla logica applicativa, all'aggiornamento dello stato dei dispositivi "polling-based" ecc.
  - *PreRendering*: è pensato per l'esecuzione di tutte quelle operazioni che, pur non essendo relative al rendering della scena, devono necessariamente essere eseguite all'interno del blocco *SceneBegi*n / *SceneEnd*. Esempi tipici sono le funzioni *gluProject* e *gluUnproject*.

- *Consente di organizzare l'applicazione in stati:*
  Il framework consente di organizzare l'applicazione in stati. Ad ogni stato viene associato un insieme (tipicamente distinto) di oggetti disegnabili, di metafore, di controller e (se necessario) di dispositivi. Ogni stato può ulteriormente essere scomposto in sotto-stati, detti "*modalità di interazione*". Approfondiremo questo argomento nel paragrafo 2.2.4.

La classe *IFApplication* è una classe astratta. Per creare una nuova applicazione occorre innanzitutto estendere tale classe e definire per essa un metodo *Init*; esso corrisponde alla funzione *OnInit* di XVR o, più in generale, alla funzione *main* di molti linguaggi di programmazione. All'interno di *Init* devono essere istanziati e legati tra loro (in genere registrandoli come *listener*) tutti gli oggetti che costituiscono l'applicazione (fonti di eventi, controller, metafore, nodi dello scene graph ecc...).

Durante l'esecuzione dell'applicazione deve esistere una ed una sola istanza della "main class" e, poiché questa fornisce i servizi di base del framework, deve essere accessibile globalmente (si tratta quindi di un **singleton**). Per tale ragione "l'oggetto Applicazione" viene istanziato automaticamente prima dell'invocazione del metodo *Init* ed assegnato alla variabile globale IFAPP, rendendolo quindi accessibile da qualsiasi punto del codice.

Si noti che, essendo la "main class" una sottoclasse di *IFApplication*, è possibile adattarne il comportamento alle esigenze specifiche della nostra applicazione semplicemente ridefinendo alcuni metodi. Tipicamente ad essere ridefiniti sono i metodi corrispondenti alle callback predefinite di XVR[3] ed in particolare il metodo *OnFrame*. Maggiori dettagli su questo aspetto ed in generale sulla classe *IFApplication* verranno forniti nel paragrafo 3.2.

2.2.3 Un primo esempio

Per riassumere quanto detto sinora e cercare di chiarirne i concetti, analizziamo un semplice esempio che mostra una implementazione alternativa del template a pagina 33, nella quale si fa uso del framework di interazione:

Listing 2.1: Primo programma con il framework di interazione

```
1 /* Nome della main class */
2 #define IFAPPNAME MyApp
3
4 /* ID associato allo stato predefinito */
5 #define IFC_APP_DEFAULT_STATE_ID    "default"
6 /* ID associato alla modalità di interazione predefinita */
7 #define IFC_CTRL_DEFAULT_INTERMOD_ID   "default"
```

[3]Questi metodi vengono invocati automaticamente e in maniera completamente trasparente al programmatore.

```
8
9  /* Inclusione dei sorgenti del framework */
10 #include "IF.h.s3d"
11
12 /* Griglia da disegnare nella scena */
13 class Grid {
14    Draw();
15
16    var _color;
17    var _size;
18 };
19
20 function Grid::Grid(color, size) { [...] }
21 function Grid::Draw() { [...] }
22
23 /* La classe "Applicazione" */
24
25 function OnDownload(userParam) { }
26
27 class IFAPPNAME:IFApplication {
28    Init(userParam);
29 };
30
31 function IFAPPNAME::Init(userParam)
32 {
33    // Creazione della metafora
34    var metafora = IFFlyingVehicle("FlyingVehicle", true);
35    metafora.SetPosition([0.0,2.0,0.0]);
36
37    // Creazione del mouse
38    var mouse = IFMouse("mouse", true);
39
40    // Creazione della tecnica di interazione
41    // per la coppia <Mouse, Flying vehicle>
42    var mouseController = IFFlyingMouseController(mouse, true, metafora,
          [...]);
43
44    // Si occupa di aggiornare la metafora ad ogni frame
45    var updateController = IFMetaphorUpdateController({metafora}, true);
46
47    // Creazione della griglia
48    var griglia = Grid([0.5, 0.5, 0.5], 100);
49
50    // Registra la griglia come listener
51    // dell'evento "Rendering"
52    this.AddRenderingListener(griglia);
53 }
```

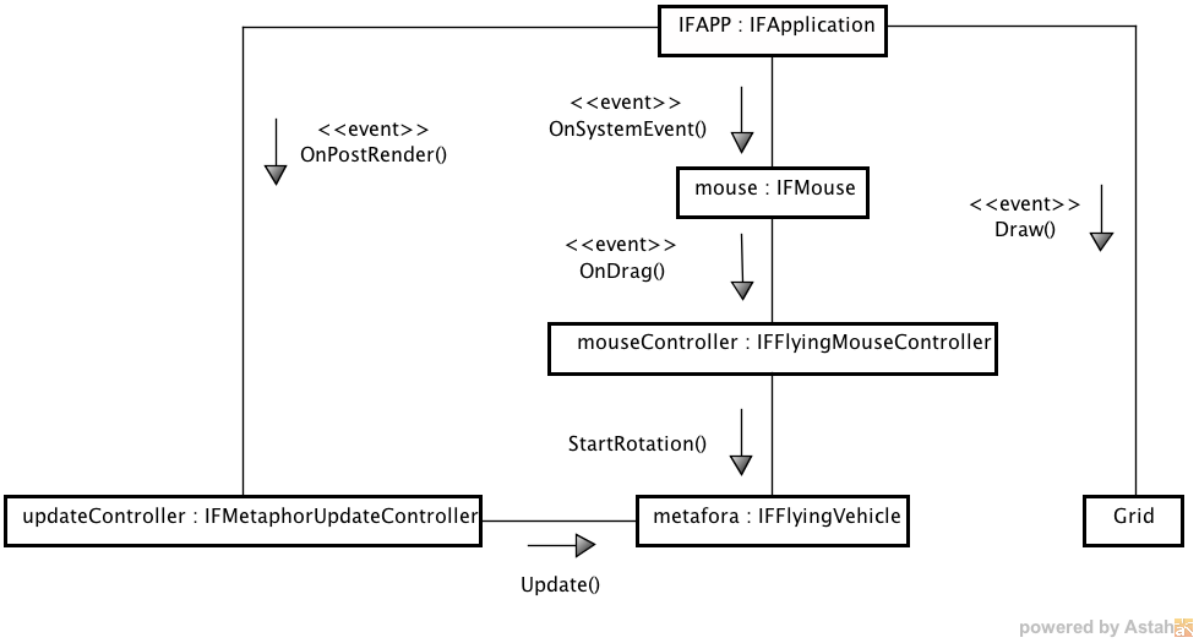
Per prima cosa viene definita la macro IFAPPNAME, tramite cui viene specificato il nome della "main class" (ovvero della classe che estende *IFApplication*). Questo è necessario per consentire al framework di istanziare tale classe. Le due costanti successive (righe 5 e 7) rappresentano gli identificatori associati allo stato predefinito dell'applicazione e alla modalità di interazione predefinita. Verranno trattate nel paragrafo 2.2.4 e per il momento possono essere ignorate. A questo punto è possibile importare i sorgenti del framework includendo il file "IF.h.s3d". Segue la definizione di ulteriori classi e funzioni e/o l'inclusione dei relativi sorgenti. Nell'esempio viene definita la sola classe "Grid", che si occupa di disegnare una griglia nella scena.

Non resta che definire la funzione *OnDownload()* (in maniera identica ai programmi S3D tradizionali) e la classe principale del programma. Come detto il nome di tale classe deve essere definito per mezzo della macro IFAPPNAME (riga 2). L'implementazione della "main class" deve prevedere (quantomeno) il metodo *Init*, contenente il codice di inizializzazione. È importante utilizzare il metodo *Init* per questo scopo e non il costruttore, in quanto molte classi del framework, al momento della loro inizializzazione, assumono che la "main class" sia già stata istanziata e assegnata alla variabile globale IFAPP. L'implementazione tipica di questo metodo segue la struttura mostrata nell'esempio:

1. Si creano e si configurano una o più metafore.
2. Si creano uno o più oggetti che rappresentano dispositivi di input.
3. Si crea una tecnica di interazione per ogni coppia <metafora, dispositivo>.
4. Si creano gli oggetti relativi al modello dell'applicazione ed i controller definiti dal programmatore (non presenti nell'esempio).
5. Si crea la scena tridimensionale e la si registra come listener dell'evento *Rendering*. Tipicamente la scena viene descritta per mezzo di uno *scene graph* che, in XVR, è costituito da una o più gerarchie di oggetti i cui i nodi interni sono oggetti di tipo *CVmObj*, mentre i nodi foglia possono essere mesh, billboard ecc. Per disegnare lo *scene graph* basta quindi registrare il nodo radice come gestore dell'evento *Rendering*. È anche possibile, come avviene nell'esempio, realizzare oggetti con un codice di disegno personalizzato che non fa uso dello *scene graph*, l'importante è che tutti gli oggetti disegnabili implementino il metodo *Draw()*.

Di seguito viene mostrato il diagramma di comunicazione del nostro semplice programma di esempio, in cui è possibile osservare come gli oggetti sono collegati tra loro, il flusso degli eventi generati ed i metodi invocati di conseguenza[4].

[4]Ovviamente non vengono mostrati tutti gli eventi generati durante l'esecuzione dell'applicazione ma, a titolo di esempio, solo alcuni tra i più significativi.



2.2.4 Stati e modalità di interazione

Consideriamo una applicazione che consente di visitare un museo virtuale, in cui è possibile selezionare le opere esposte per ottenere informazioni su di esse. In una possibile implementazione possiamo individuare almeno tre stati significativi:

- Lo stato principale, nel quale l'utente naviga all'interno del museo e può selezionare le opere esposte.
- Uno stato nel quale viene visualizzata la sola opera selezionata ed in cui l'utente ha la possibilità di esaminarla da vicino attraverso una apposita metafora (es. l'*Orbiting*). Viene inoltre mostrata una descrizione dettagliata dell'opera e della sua storia.
- Una schermata iniziale, comprensiva di un pannello di configurazione.

Ad ognuno di questi stati corrisponderà un diverso insieme di oggetti da disegnare a schermo, di controller, ed eventualmente di dispositivi di input. Da quanto detto tali oggetti sono tutti listener dell'oggetto "applicazione", quindi uno stato dell'applicazione può essere definito come un insieme di listener registrati presso IFApplication[5].

Ad ogni stato è associato un identificatore unico, che può essere di tipo stringa o intero. L'identificatore dello stato di default deve essere dichiarato esplicitamente, definendo la costante **IFC_APP_DEFAULT_STATE_ID** all'inizio del programma (come mostrato nel listato 2.1, riga 5). Ovviamente definendo questo identificatore se ne dichiara

[5]I dettagli sulla composizione degli stati della classe *IFApplication* verranno forniti nel paragrafo 3.2

implicitamente anche il tipo (stringa o intero), quindi gli identificatori degli altri stati dovranno essere coerenti con questa scelta.

I principali metodi che la classe IFApplication definisce per la gestione degli stati sono i seguenti:

CreateState - Crea un nuovo stato (senza modificare lo stato corrente).

RemoveState - Elimina lo stato specificato (se si cerca di eliminare lo stato corrente, la rimozione non ha effetto).

ChangeState - Effettua un cambiamento di stato.

Una volta richiesto un cambiamento di stato, ma prima che questo venga portato a termine, viene generato l'evento *StateWillChange*. Tale evento ha il fine di dare la possibilità agli oggetti interessati di prepararsi all'imminente cambio di stato, ad esempio effettuando operazioni di "*cleanup*" e resettando il proprio stato interno.

La creazione e l'inizializzazione degli stati avviene tipicamente nella *OnInit*, seguendo la seguente procedura:

1. Si crea un nuovo stato per mezzo del metodo *CreateState*.
2. Si passa allo stato appena creato invocando il metodo *ChangeState*.
3. Si aggiungono al nuovo stato corrente tutti gli elementi che lo compongono[6].
4. Si ripete questa procedura per ognuno degli stati rimanenti.
5. Infine si ripristina lo stato di default come stato corrente per mezzo della seguente chiamata:
   ChangeState(IFC_APP_DEFAULT_STATE_ID)

Il listato seguente illustra questa la procedura:

```
/* Nome della main class */
#define IFAPPNAME EsempioStati

// ID degli stati dell'applicazione
// In questo esempio si è scelto di usare identificatori interi
#define IFC_APP_DEFAULT_STATE_ID  0
#define IFC_APP_STATE1_ID         1
#define IFC_APP_STATE2_ID         1
```

[6]Quando viene registrato un listener, esso viene aggiunto allo stato corrente.

```
[...]

class IFAPPNAME:IFApplication {
    Init(userParam);
    [...]
};

function IFAPPNAME::Init(userParam)
{
    [...]

    // Creazione dello stato 1
    this.CreateState(IFC_APP_STATE1_ID);
    this.ChangeState(IFC_APP_STATE1_ID, false);
    // Aggiunta degli elementi dello stato 1
    [...]

    // Creazione dello stato 2
    this.CreateState(IFC_APP_STATE2_ID);
    this.ChangeState(IFC_APP_STATE2_ID, false);
    // Aggiunta degli elementi dello stato 2
    [...]

    // Ripristino e inizializzo lo stato iniziale
    this.ChangeState(IFC_APP_DEFAULT_STATE_ID, false);
    // Aggiunta degli elementi dello stato 0
    [...]
}
```

Si noti il secondo parametro del metodo *ChangeState*. Esso specifica se deve essere inviato l'evento *StateWillChange* prima dell'effettivo cambiamento di stato. In fase di inizializzazione questo parametro deve essere sempre settato a *false* (oppure omesso, è un parametro opzionale). Al contrario, ad ogni successiva invocazione questo parametro dovrebbe essere sempre settato a *true*, al fine di dare la possibilità agli oggetti interessati di prepararsi all'imminente cambiamento di stato.

All'interno di uno stato è possibile definire più sotto-stati, detti "***modalità di interazione***". Il nome deriva dal fatto che, passando da una modalità all'altra, l'utente può interagire in maniera differente con l'ambiente virtuale. Modalità di interazione differenti si distinguono tra loro in funzione di:

- La metafora di interazione in uso (per eseguire task differenti o per eseguire uno stesso task in modi diversi).
- L'uso di *tecniche di interazione* alternative.

- L'impiego di un differente insieme di dispositivi.
- Ecc...

In termini implementativi questo si traduce nella realizzazione delle modalità di interazione come **stati degli oggetti controller** (anziché stati della classe *IFApplication*). In altri termini, gli oggetti controller modificano il loro comportamento (cioè le azioni da essi intraprese in risposta agli eventi) al variare della modalità di interazione corrente. Ovviamente, a seconda della propria funzione, un controller può ignorare completamente questo meccanismo, oppure supportare soltanto un sottoinsieme delle modalità usate dall'applicazione: quando ad un controller viene notificata l'attivazione di una modalità da lui non supportata, esso si limita ad ignorare la notifica, mantenendo il proprio stato corrente. Da quanto detto segue che i metodi per la gestione delle modalità di interazione sono forniti dalle classi controller, anziché da *IFApplication*. Esamineremo questi metodi nel paragrafo 3.4. La classe *IFApplication* si limita a fornire i meccanismi per richiedere un cambio di modalità e per inoltrare tale richiesta agli oggetti interessati: il metodo preposto a questo è *ChangeInteractionModality*, la cui implementazione consiste nella generazione di due eventi:

InteractionModalityWillChange - Ha lo scopo di preparare gli oggetti interessati all'imminente cambio di modalità.

InteractionModalityChange - Rappresenta la vera e propria richiesta di cambiamento di modalità di interazione. Alla ricezione di tale evento i controller modificheranno il loro stato interno di conseguenza.

Così come per gli stati dell'applicazione, esiste una modalità di interazione predefinita, il cui identificatore (di tipo intero o stringa) deve essere definito per mezzo della costante **IFC_CTRL_DEFAULT_INTERMOD_ID** (come mostrato nel listato 2.1, riga 7).

Nota

Una volta **conclusa la fase di inizializzazione** (*OnInit*), i comandi relativi alla creazione, alla rimozione e il cambio di stati e modalità di interazione **devono essere eseguiti sfruttando il meccanismo di esecuzione differita**, come descritto nel paragrafo 2.2.6.

2.2.5 Gestione degli eventi

Uno degli obiettivi del framework consiste nel fornire i meccanismi e le astrazioni necessari ad implementare l'interazione utente in modo semplice e coerente. Particolare attenzione è stata quindi posta al fine di individuare un design che garantisse una gestione uniforme

dell'input (proveniente da dispositivi di varia natura) e delle notifiche interne all'applicazione stessa. Il paradigma *Event-driven* ed i pattern ad esso correlati si sono rivelati la soluzione ideale.

Uno dei pattern più usati in questo paradigma è l'*Observer*, detto anche *Publish-Subscribe* o *Delegation Event Model*[7] [16, 23]. Tale pattern prevede:

- Una **sorgente di eventi** (detta anche *publisher*)
- Dei **consumatori di eventi**[8] (detti anche listener, observer o subscriber)
- Un'interfaccia che specifica la modalità attraverso cui la sorgente notifica l'evento ai consumatori.

La sorgente definisce gli eventi da essa prodotti e gestisce una struttura dati contenente i riferimenti ai listener (tipicamente un array dinamico o una lista concatenata), fornendo i metodi per l'aggiunta e la rimozione dei listener. La generazione di un evento (che in genere corrisponde ad una variazione dello stato interno della sorgente), e quindi la notifica dello stesso, consiste nell'invocazione di un dato metodo su tutti i consumatori registrati. Per tale ragione tutti i consumatori devono implementare un'interfaccia nella quale vengono dichiarati i metodi (detti *event handler*) invocati dalla sorgente al verificarsi dei rispettivi eventi.

Il seguente diagramma delle classi illustra quanto detto:

[7] Questo nome è stata introdotto dalla Sun Microsystems con la release 1.1 di Java, nella quale è stato adottato un nuovo modello di gestione degli eventi per l'Abstract Window Toolkit (AWT). Per maggiori dettagli si visiti la pagina http://docs.oracle.com/javase/1.4.2/docs/guide/awt/1.3/designspec/events.html

[8] Tipicamente sono oggetti "*controller*", ma non necessariamente: gli oggetti che astraggono dispositivi di input, ad esempio, sono sia sorgenti che consumatori di eventi. Il loro compito è infatti quello di convertire gli eventi ricevuti da *IFApplication* in eventi di livello più alto.

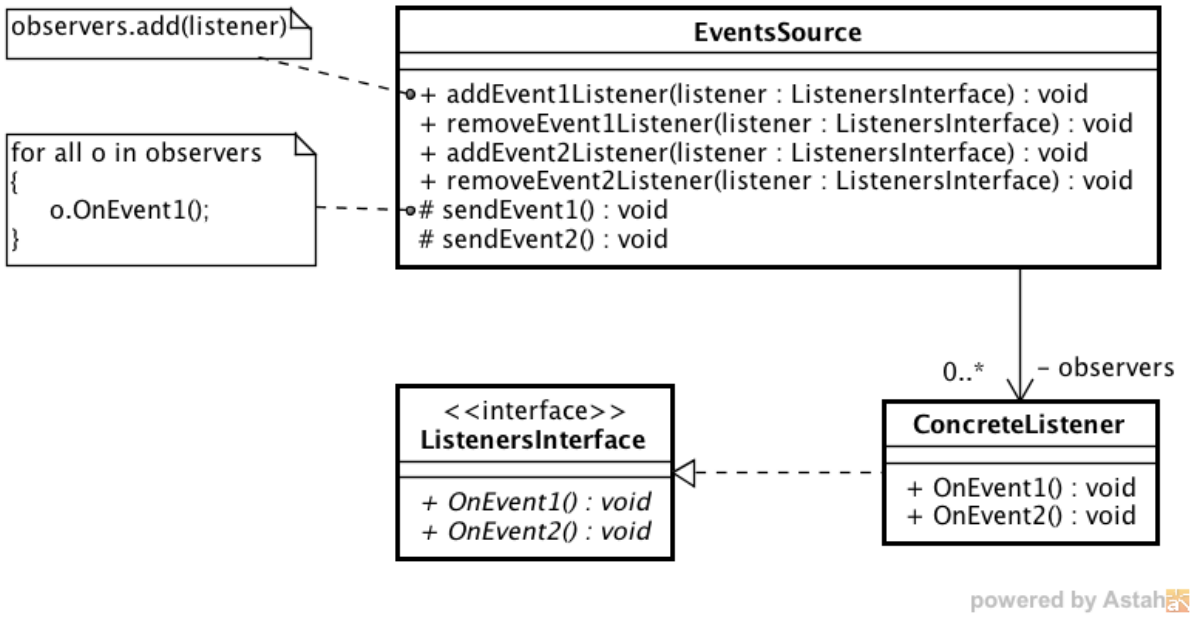


Figura 2.1: Il pattern Observer

Nel framework le interfacce dei listener vengono specificate nella documentazione relativa alle sorgenti. Questo è dovuto al fatto che il linguaggio S3D non supporta tale costrutto. In alternativa si sarebbero potute definire delle classi "*Adapter*"[9], ovvero classi che forniscono un'implementazione di default (vuota) di ogni event handler, e che devono essere "sotto-classate" da ogni listener. Tuttavia si è preferito adottare un approccio diverso, nel quale i listener di ogni evento vengono gestiti separatamente[10]. Segue che ogni listener si registra presso la sorgente per i soli eventi a cui è interessato e di conseguenza solo quelli gli verranno notificati. Questo approccio ha il vantaggio di minimizzare il numero di chiamate di metodo, in quanto ad ogni oggetto vengono notificati i soli eventi che può (e vuole) effettivamente gestire.

Dato un evento EVT, i nomi metodi per la gestione dei listener ed i nomi degli *event handler* seguono la seguente convenzione:

- Il metodo per la registrazione di un listener ha una firma del tipo:
  *Add*EVT*Listener(listener)*
- Il metodo per la de-registrazione di un listener ha una firma del tipo:
  *Remove*EVT*Listener(listener)*
- Il metodo di gestione dell'evento deve avere una delle seguenti firme:

[9] Come avviene nelle librerie AWT e Swing di Java.
[10] Salvo rare eccezione che verranno evidenziate nei capitoli successivi.

- *On*Evt()
- *On*Evt(*descriptor*) - dove *descriptor* è un descrittore contenente informazioni aggiuntive sull'evento

Consideriamo ora il seguente esempio, in cui un oggetto di tipo *IFMouse* viene registrato presso *IFApplication* per ricevere gli eventi di cui necessita:

```
// Creo un oggetto di tipo IFMouse.
// Il primo parametro del costruttore è l'id dell'oggetto,
// il secondo verrà discusso a breve e può essere ignorato per ora.
var mouse = IFMouse("mouse", false);

// Registrazione del mouse presso IFApplication
IFAPP.AddPostRenderingListener(mouse);
IFAPP.AddSystemEventFiredListener(mouse);
```

Questo codice è perfettamente valido, tuttavia è abbastanza tedioso per il programmatore dover consultare la documentazione per scoprire gli eventi di cui necessita *IFMouse*. Inoltre il dimenticarsi anche di una sola registrazione produce bug di difficile individuazione. Per tale ragione ogni oggetto che deve "mettersi in ascolto" di uno o più eventi deve fornire due metodi, *Register()* e *Unregister()*, che provvedono ad effettuare tutte le registrazioni / de-registrazioni necessarie. Il codice precedente può allora essere riscritto come segue:

```
var mouse = IFMouse("mouse", false);
mouse.Register();
```

Poiché è molto comune registrare un oggetto subito dopo averlo creato, è stato previsto un apposito parametro booleano nel costruttore che attiva automaticamente la registrazione dopo l'inizializzazione dell'oggetto stesso. Sfruttando questo parametro il listato precedente si riduce ad una sola riga di codice:

```
var mouse = IFMouse("mouse", true);
```

Nota

Una volta **conclusa la fase di inizializzazione** (*OnInit*), i comandi relativi all'aggiunta o alla rimozione di listener (compresa l'invocazione dei metodi *Register()* e *Unregister()*) **devono essere eseguiti sfruttando il meccanismo di esecuzione differita**, come descritto nel paragrafo 2.2.6.

Si noti che i consumatori devono, in un qualche modo, ottenere il riferimento alla sorgente di eventi. Questo è necessario sia per effettuare le operazioni di registrazione e de-registrazione, sia quando il consumatore ha bisogno di interrogare lo stato della sorgente (in genere durante la gestione di un evento). Tradizionalmente questo problema viene risolto in due modi:

- Il riferimento alla sorgente viene passato come parametro al metodo di gestione dell'evento (direttamente o come campo del descrittore).
- Si memorizza il riferimento alla sorgente in una variabile di istanza del consumatore.

La prima soluzione non consente però di ottenere il riferimento al di fuori del gestore dell'evento. Quanto alla seconda dobbiamo tener presente che, poiché il linguaggio S3D effettua una garbage collection basata su **reference counting**, la presenza di **riferimenti ciclici** (come quelli prodotti da questo approccio) comporta un *memory leak*. Questo è un problema da tener ben presente quando si scrive un programma S3D, ed ha influenzato non poche scelte nella progettazione del framework.

Ovviamente questo problema non si pone quando la sorgente in questione è l'oggetto "Applicazione". Infatti, come spiegato in precedenza, il suo riferimento è memorizzato nella variabile globale IFAPP, ed è quindi accessibile da qualsiasi punto del codice. Per quanto riguarda le altre sorgenti, la soluzione adottata consiste nell'inserire nello stato dell'applicazione una tabella hash in cui vengono memorizzati i riferimenti a tutte le fonti di eventi dello stato corrente. A tal scopo la classe IFApplication fornisce i metodi necessari per la gestione della tabella. L'inserimento in tabella di una fonte di eventi viene effettuato dal metodo *Register()* della fonte stessa, rendendo questa procedura completamente trasparente al programmatore. Con questo approccio i listener devono quindi memorizzare soltanto l'identificatore delle sorgenti a cui sono registrati, e poi reperire il riferimento corrispondente attraverso il metodo *IFApplication::GetEventsSourceById()*.

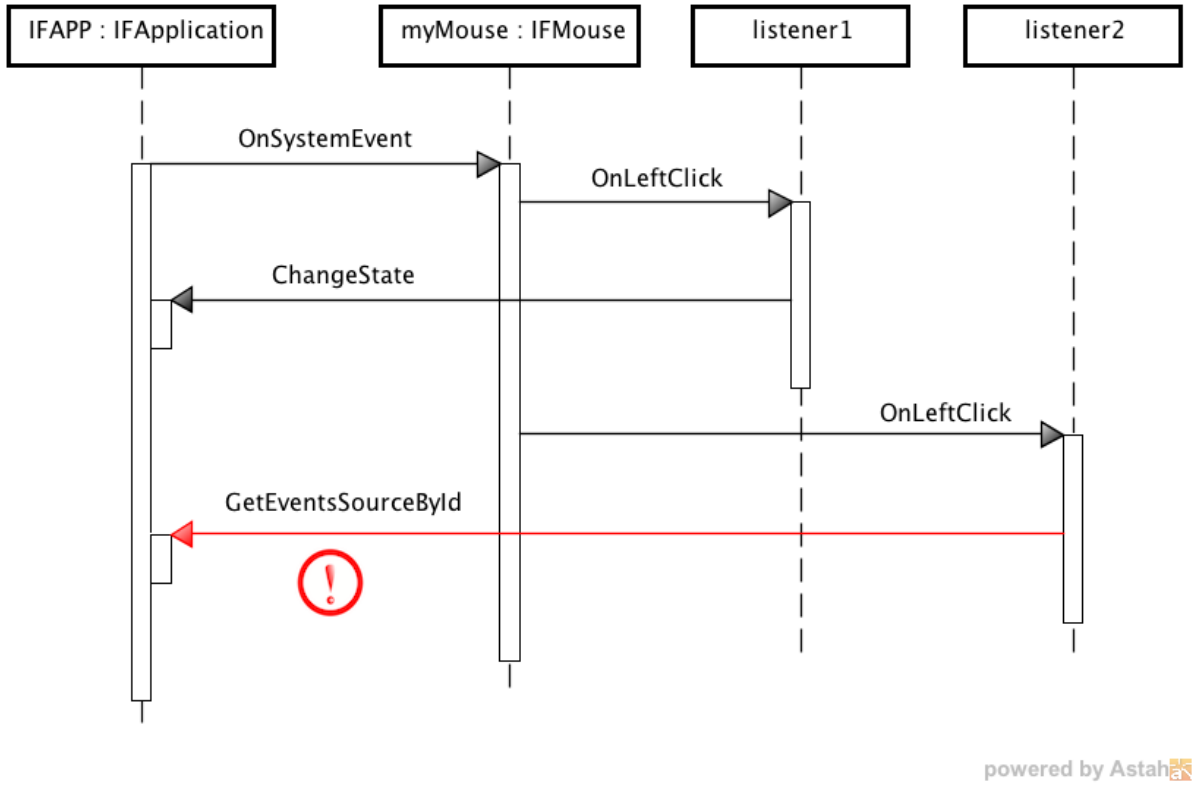
Un'ultima considerazione: gli eventi vengono notificati ai listener seguendo lo stesso ordine con cui sono stati registrati. Per tale ragione i listener dovrebbero essere implementati facendo attenzione a non creare dipendenze logiche tra di loro. Esistono però casi in cui l'ordinamento dei listener è indispensabile, come nel caso dell'evento *Rendering* in presenza di elementi trasparenti o bidimensionali.

2.2.6 Esecuzione differita

Ci sono categorie di operazioni che, se eseguite direttamente o indirettamente durante la gestione di un evento, possono produrre bug di difficile comprensione ed individuazione. Per chiarire il problema, di seguito vengono illustrati due possibili scenari in cui questo accade.

Esempio 1

Supponiamo di avere un oggetto di tipo *IFMouse* (chiamato *myMouse*) e due listener per l'evento *LeftClick* (chiamati *listener1* e *listener2*). Il primo listener, una volta ricevuto l'evento ed in presenza di opportune condizioni, esegue un cambiamento di stato. Il secondo invece, per svolgere il compito a lui assegnato, deve interrogare lo stato del mouse. Come spiegato in precedenza, *listener2* non possiede un riferimento *myMouse* ma può procurarselo attraverso il metodo *GetEventsSourceById* della classe *IFApplication*. Se *listener1* riceve per primo l'evento (ovvero se *listener1* è stato registrato prima di *listener2*), lo stato dell'applicazione cambierà prima che l'evento venga notificato a *listener2*. Segue che *listener2* non riuscirà ad ottenere il riferimento a *myMouse*.

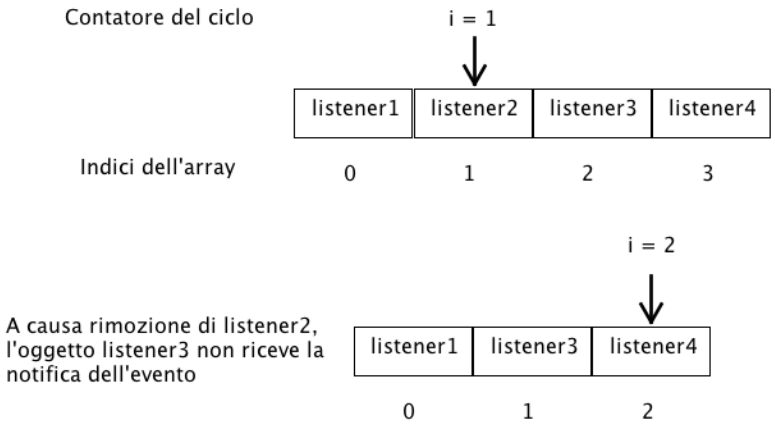


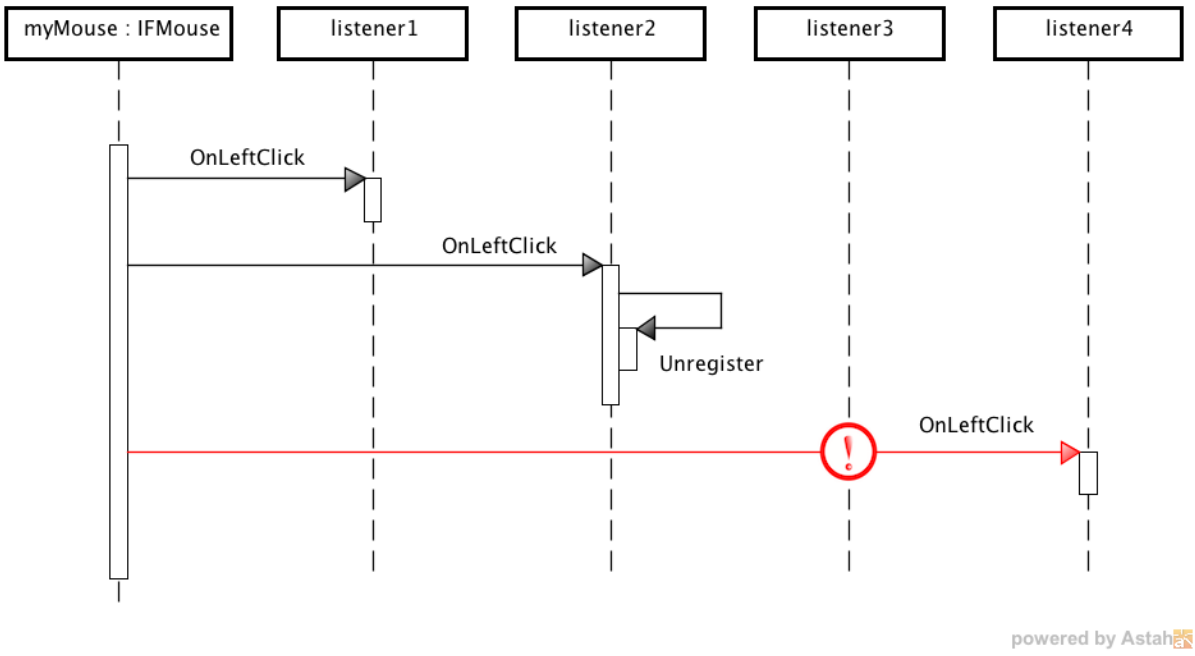
Si tratta di un caso di dipendenza logica tra i due listener. Ovviamente in questo caso il problema potrebbe essere risolto invertendo l'ordine di registrazione, tuttavia è auspicabile trovare una soluzione più generale e affidabile.

Esempio 2

Supponiamo di avere un oggetto di tipo *IFMouse* e quattro listener per l'evento *LeftClick*. Il listener2, una volta ricevuto l'evento ed in presenza di opportune condizioni, svolge i compiti a lui assegnati, si de-registra, e termina il suo ciclo di vita. Le operazioni eseguite dagli altri listener non sono significative per questo esempio. La classe IFMouse inserisce

i riferimenti ai consumatori in un array dinamico. La notifica di un evento consiste in un ciclo *for* nel quale si scorre l'array invocando il metodo *OnLeftClick* su ogni elemento. Terminata la gestione dell'evento da parte di *listener2*, il contatore del ciclo for viene incrementato assumendo come valore 2. Secondo le aspettative del programmatore, tale indice corrisponde a *listener3.* Tuttavia, a causa della de-registrazione di *listener2*, quell'elemento dell'array è ora occupato da *listener4*. Segue che *listener3* non verrà notificato del verificarsi dell'evento.





Questo è uno dei classici bug dovuti alla modifica di una collezione di oggetti mentre si sta iterando su di essa.

In generale, questi problemi possono verificarsi quando, **una volta terminata la fase di inizializzazione**, viene eseguito un comando appartenente ad una delle seguenti categorie:

- **Aggiunta o rimozione di listener**, compresi i metodi *Register()*[11] e *Unregister()*.
- **Creazione o rimozione di uno stato**.
- **Cambiamento di stato**.
- **Creazione o rimozione di una modalità di interazione**.
- **Cambiamento della modalità di interazione**.

Una possibile soluzione del problema consiste nel differire l'esecuzione di questi comandi al termine della fase di gestione degli eventi. Dal punto di vista implementativo, è possibile realizzare questa funzionalità sfruttando il pattern *Command Processor* [5], che consente di separate la richiesta di un servizio dalla sua esecuzione. Essendo basato sul pattern *Command* [16], anche questo pattern prevede che le richieste per determinati servizi vengano incapsulate all'interno di oggetti, detti "*command*". La principale differenza tra i due deriva però dalla presenza di un oggetto "*command processor*" incaricato dello scheduling e dell'esecuzione delle richieste. I partecipanti previsti dal pattern sono:

Abstract Command - Classe astratta nella quale viene definita l'interfaccia per l'esecuzione dei "*command*".

Command - Rientrano in questa categoria tutte le classi che estendono AbstractCommand. Ognuna di queste classi rappresenta una specifica richiesta ed implementa i metodi necessari alla sua esecuzione.

Command Processor - Gestisce una collezione di oggetti "*command*" e ne attiva l'esecuzione secondo le politiche stabilite dalla classe stessa.

Controller - In risposta ad uno "stimolo" esterno istanzia un opportuno oggetto "*command*" e ne delega l'esecuzione al "command processor". Dal punto di vista implementativo lo "stimolo" altro non è che l'invocazione di un suo metodo, anche se concettualmente esso può rappresentare la richiesta di un servizio da parte di un'altra componente dell'applicazione oppure la ricezione di una notifica di qualche tipo (come avviene in questo framework).

[11] Anche quando invocato automaticamente dal costruttore di un oggetto.

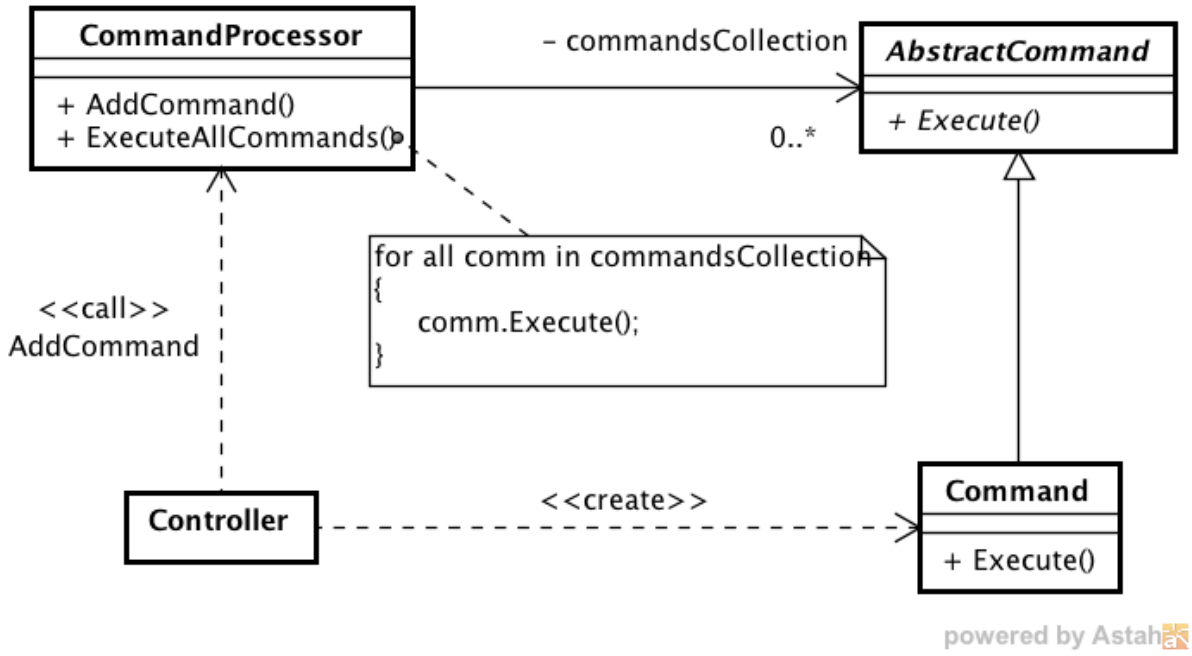


Figura 2.2: Il pattern Command Processor

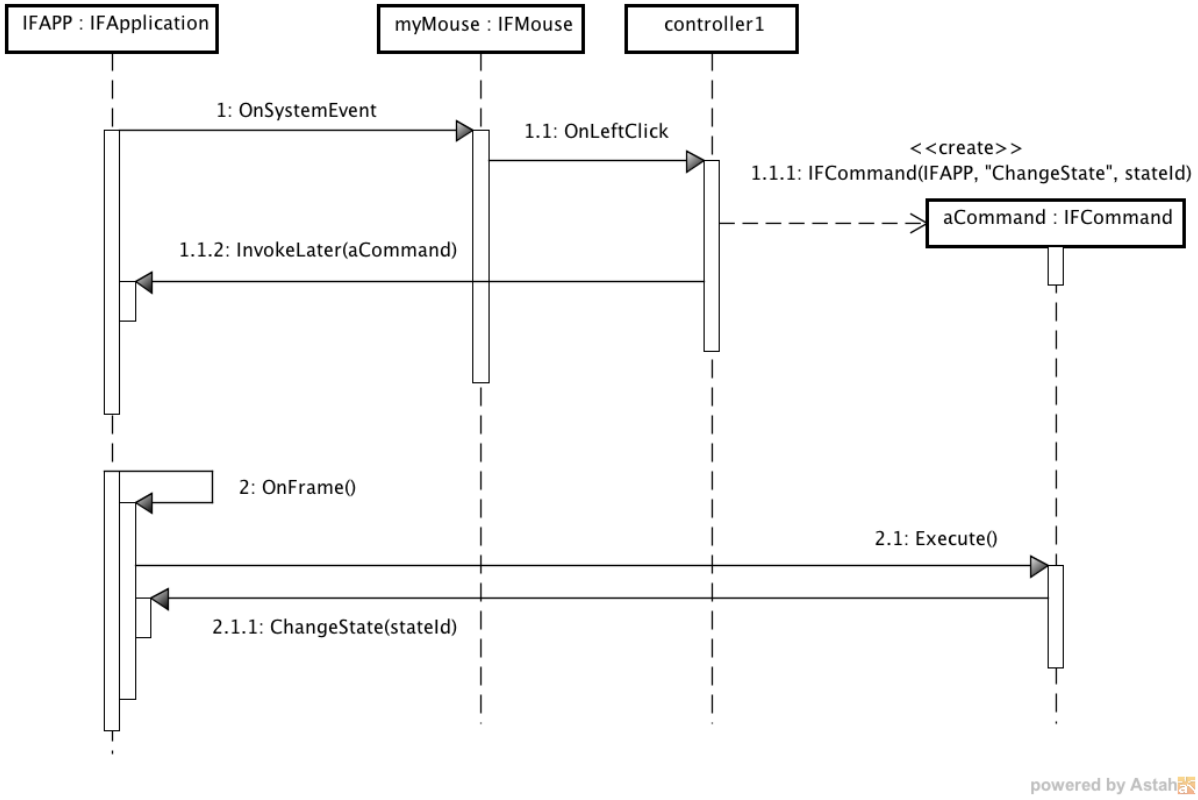
Vediamo ora come sono stati distribuiti questi ruoli tra le classi del framework.
La classe *AbstractCommand* non è stata definita esplicitamente, tuttavia essa consiste nel solo metodo *Execute()*, che deve essere implementato da tutte le classi "*command*". In genere queste classi vengono definite dal programmatore in funzione delle specifiche necessità. Nel caso in cui tutto quello che si vuole fare consiste nell'invocazione di un singolo metodo, è possibile usare la classe di utilità *IFCommand*. L'interfaccia di tale classe è costituita da un costruttore e dal metodo *Execute*. Quest'ultimo si limita ad invocare un certo metodo su un dato oggetto, passando gli opportuni parametri. Come è facile intuire, il costruttore prende come parametro un riferimento all'oggetto sul quale invocare il metodo, il nome del metodo stesso (come stringa) ed una lista di parametri da passare al metodo.

Il ruolo di "*command processor*" è svolto dalla classe *IFApplication*. Essa gestisce una coda di "*command*" e fornisce il metodo *InvokeLater* per aggiungere un comando alla coda. L'esecuzione dei comandi consiste semplicemente in un ciclo nel quale, ad ogni iterazione, viene rimosso l'elemento in testa alla coda e viene invocato il metodo *Execute* su di esso. Come accennato in precedenza, l'esecuzione di questi comandi deve avvenire periodicamente e al termine della fase di gestione degli eventi. In XVR questo si traduce nell'ultima parte del ciclo *OnFrame()*. Si noti infine che, per evitare bug come quello illustrato nell'esempio 1, la coda è unica e indipendente da stati e modalità di interazione.

Quanto ai "*controller*", come si può intuire, si tratta di un sottoinsieme degli oggetti controller descritti nel paragrafo 2.2.1. Alla ricezione di un dato evento, essi creano un

oggetto "*command*" di tipo appropriato e ne delegano l'esecuzione alla classe *IFApplication* per mezzo del metodo *InvokeLater*.

Il seguente diagramma di attività fornisce un esempio di quanto detto:



2.2.7 Architettura del framework di interazione

In questo paragrafo esamineremo l'architettura del framework, i moduli che lo compongono e le loro caratteristiche.

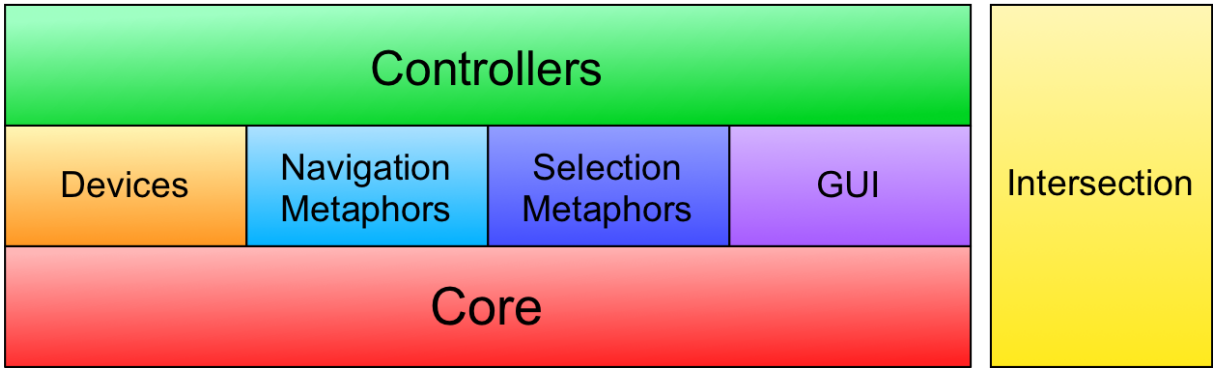


Figura 2.3: Architettura del framework

Le classi che costituiscono il framework sono concettualmente organizzate in nove *package*[12], in accordo alle funzionalità offerte.

Alla base di tutto troviamo il package "**Core**", che comprende la classe *IFApplication* e le classi astratte relative alle fonti di eventi ed ai controller. Queste classi verranno descritte in dettaglio nel capitolo 3.

Il package "**Devices**" è costituito dalle classi associate alle fonti di eventi di input (tipicamente dispositivi). Il principale scopo di queste classi è quello di convertire gli eventi ricevuti da *IFApplication* in eventi alto livello tipici del dispositivo che esse astraggono. Allo stato attuale i dispositivi di input supportati sono il mouse, la tastiera, il touchscreen ed il Microsoft Kinect. È inoltre presente una classe che consente lo scambio di messaggi tra il player XVR ed il suo contenitore. Maggiori dettagli verranno forniti nel capitolo 4.

I package "**NavigationMetaphors**" e "**SelectionMetaphors**" contengono rispettivamente le metafore per la navigazione e per la selezione. Queste rappresentano una selezione delle metafore illustrate nel capitolo 1. In particolare, sono state scelte alcune tra le metafore più usate nell'ambito delle applicazioni per la fruizione di contenuti culturali. Il compito di manipolare le metafore in risposta all'input dell'utente (ovvero il compito di implementare una specifica *tecnica di interazione*) è affidato ai rispettivi controller, definiti nei sotto-package "**Controllers**". In genere è presente una classe controller per ogni coppia <dispositivo, metafora>, salvo i casi in cui un dispositivo è palesemente inadatto per controllare una certa metafora oppure quando questo, sebbene possibile, sarebbe stato privo di qualunque utilità. La tabella seguente mostra le metafore disponibili ed i dispositivi con i quali possono essere usate. Le classi che implementano queste metafore ed i relativi controller verranno trattate nei capitoli 5 e 6.

	Metafore (parte 1)			
	Flying vehicle	*Walking*	*Speed-coupled flying*	*Orbiting*
Mouse	√	√	√	√
Tastiera	√	√	√	√
Touchscreen	√	√		√
Kinect	√	√	√	√

[12]Si noti che il linguaggio S3D non fornisce nessun meccanismo simile ai *package* o ai *namespace*. Segue che questa organizzazione è fondamentalmente concettuale e si riflette solo sul modo in cui sono stati organizzati i sorgenti e sulla terminologia usata nella documentazione.

	Metafore (parte 2)			
	Cylindrical orbiting	*Overhead crane*	*Ray selection*	*Cursor selection*
Mouse	√	√	√	√
Tastiera	√	√		
Touchscreen	√	√	√	
Kinect	√	√	√	√

Come si può notare il framework non fornisce metafore di manipolazione. Questo perché, al fine di porre dei limiti al progetto, si è deciso di concentrare gli sforzi sulle metafore maggiormente usate nell'ambito della fruizione di contenuti culturali. In questa categoria di applicazioni è abbastanza raro che l'utente possa manipolare gli elementi della scena e, quando ciò accade, si tratta quasi sempre di soluzioni ad hoc. Per tali ragioni, l'implementazione di metafore di manipolazione utilizzabili in un contesto più generale viene rinviata ad eventuali sviluppi futuri.

Il package "**GUI**", trattato nel capitolo 7, contiene le classi necessarie alla realizzazione di semplici interfacce grafiche. Esso comprende alcuni controlli grafici di base (come il *button*, il *toggle-button* e il *panel*) ed un meccanismo di *layout* sufficientemente flessibile. Una delle peculiarità di questo modulo, che lo distingue dai *toolkit* per interfacce grafiche tradizionali, consiste nella netta separazione tra il controllo grafico in sé ed i controller che ne alterano lo stato in risposta all'input dell'utente. Questo consente il supporto a dispositivi di input di qualsiasi natura. Concludendo è bene precisare che questo modulo è da considerarsi un primo tentativo di integrare nel framework le funzionalità necessarie alla realizzazione di una semplice interfaccia grafica. Sebbene funzionante e utilizzabile già allo stato attuale, alcuni suoi aspetti possono essere migliorati in futuro.

L'ultimo package, "**Intersection**", è trasversale e indipendente dal resto del framework. Illustrato nel capitolo 8, racchiude le strutture dati e gli algoritmi per effettuare test di intersezione (statici e dinamici) tra primitive geometriche. La possibilità di effettuare questo tipo di test in maniera efficiente è infatti indispensabile nella realizzazione di funzionalità quali la *collision detection* ed il *terrain following*, oltre che la quasi totalità delle metafore di selezione.

In totale, il framework espone al programmatore 88 classi pubbliche. L'intero progetto[13] è costituito da circa 27000 linee di codice.

2.2.8 Documentazione e testing

La verifica del framework è stata effettuata attraverso numerose batterie di test automatici e/o sviluppando alcune semplici applicazioni interattive (quando necessario o conveniente). Queste applicazioni, di cui vengono riportati alcuni screenshot, si sono rivelate un

[13] Compreso il codice relativo al testing.

aiuto prezioso anche per avere un feedback sulle funzionalità e sull'interfaccia esposta dal framework, evidenziando carenze o inconsistenze di progettazione.

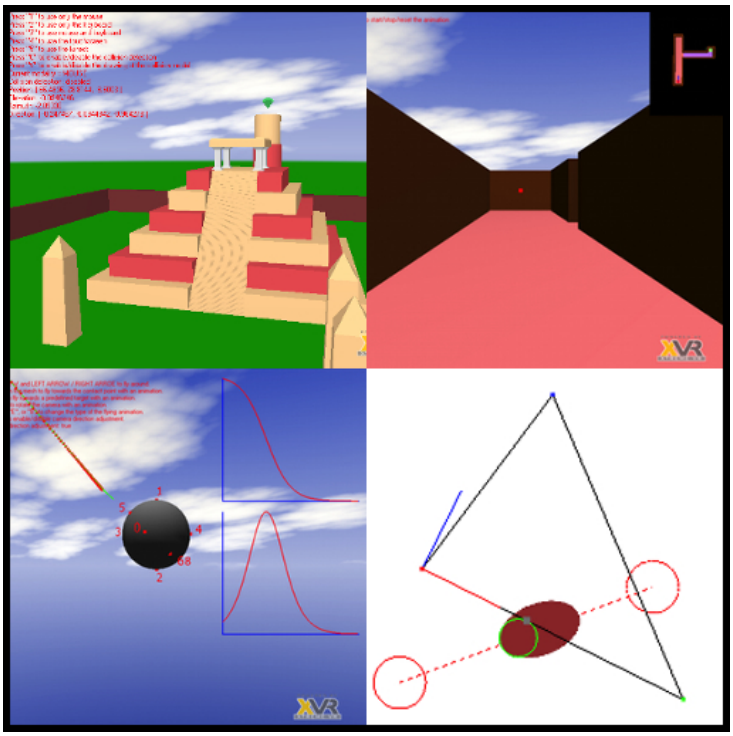


Figura 2.4: Alcune applicazioni sviluppate per testare il framework

Il framework è inoltre corredato da una corposa documentazione in lingua inglese, presente nei sorgenti ed esportata in formato HTML. La documentazione in formato HTML è stata ottenuta utilizzando lo strumento **Natural Docs**[14]**,** opportunamente configurato per adattarsi alle caratteristiche del linguaggio S3D.

[14]http://www.naturaldocs.org

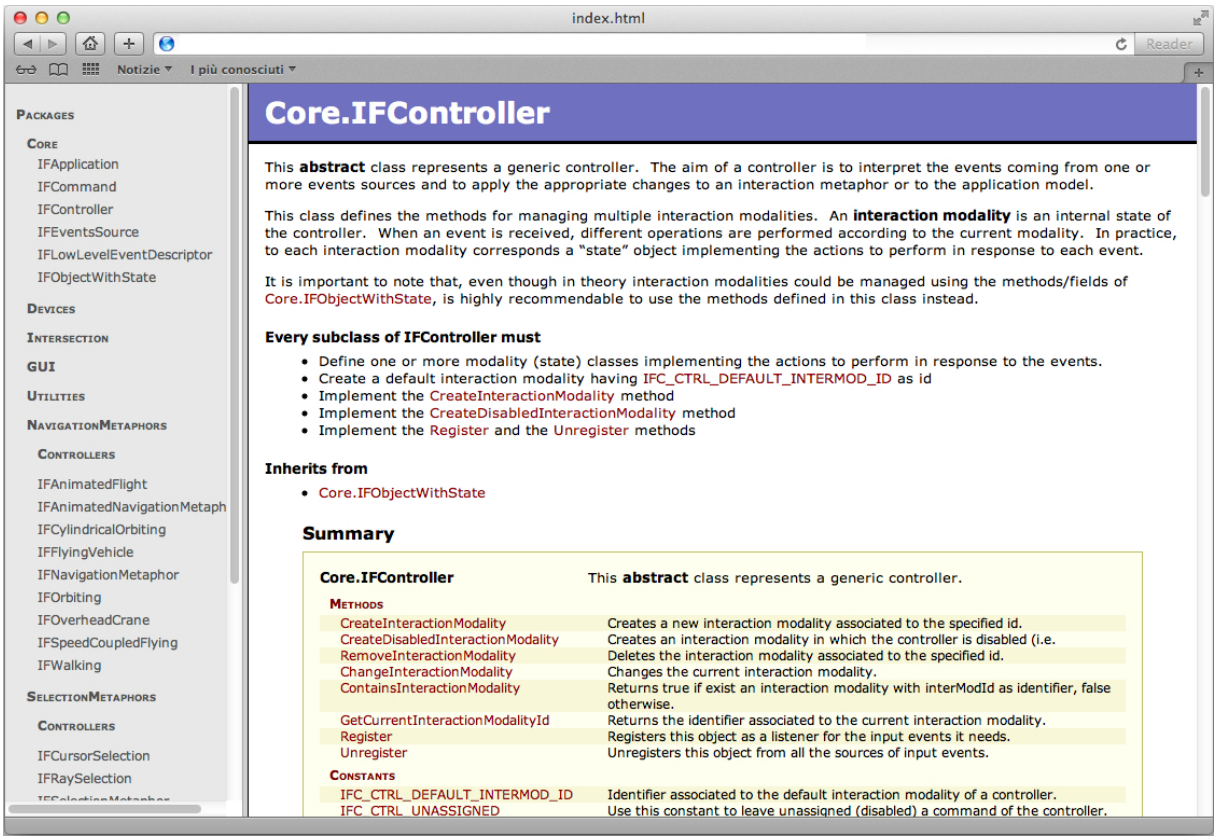


Figura 2.5: La documentazione del framework in formato HTML

Capitolo 3

Il package “Core”

3.1 La classe IFObjectWithState

È comune che una classe debba cambiare il suo comportamento (ovvero le azioni intraprese dai propri metodi) in funzione del suo stato interno. Nei casi più semplici si ricorrere tipicamente ad un'implementazione basata su blocchi di scelte condizionali. Tuttavia, al crescere del numero di stati significativi, le dimensioni di tali blocchi aumentano, compromettendone modificabilità ed estensibilità. In questi casi è utile ricorrere al pattern *State* [16], il quale prevede che tutta la logica relativa ad uno specifico stato venga incapsulata all'interno di un oggetto. I partecipanti al pattern sono i seguenti (mostrati in figura 3.1):

Context - Rappresenta una classe il cui comportamento varia in funzione del suo stato interno. Contiene un'istanza di una delle sottoclassi concrete di State, che rappresenta il suo stato corrente. Ogni richiesta che viene inoltrata a *Context* e che dipende dal suo stato interno viene delegata allo stato corrente. In figura tale richiesta è rappresentata dal metodo *Request*, mentre la delega consiste nell'invocazione del metodo *Handle* sullo stato corrente. Segue che tutti gli stati devono implementare una stessa interfaccia. La transizione da uno stato all'altro può essere decisa da *Context* oppure può essere richiesta “dall'esterno” esponendo un opportuno metodo (SetState in figura).

State - Classe astratta che definisce l'interfaccia comune ad ogni stato.

ConcreteState - Implementa uno specifico stato interno di *Context*.

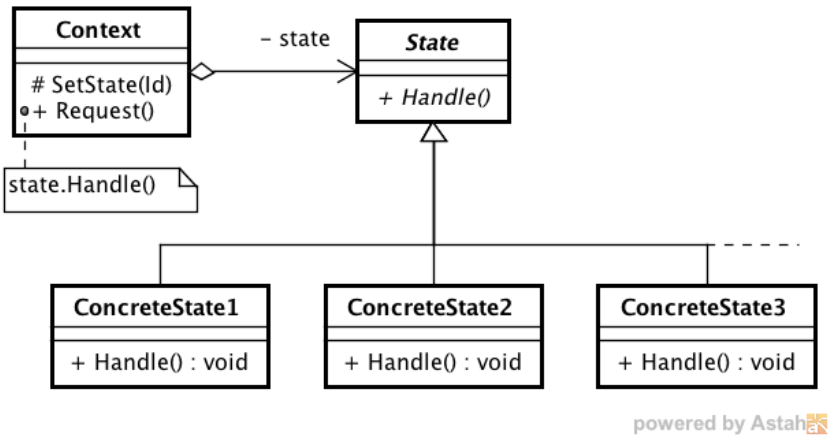


Figura 3.1: Il pattern *State*

IFObjectWithState è una classe di utilità che fornisce alle sue sottoclassi le funzionalità di base per la gestione degli stati. Concettualmente, essa implementa una parte delle funzionalità del *Context*. È bene precisare che *IFObjectWithState* non fa alcuna assunzione sul tipo degli oggetti stato che va a gestire, quindi il programmatore può e deve definire tali classi come ritiene più opportuno. Ogni istanza di stato viene associata ad un identificatore unico, che può essere un intero o una stringa[1].

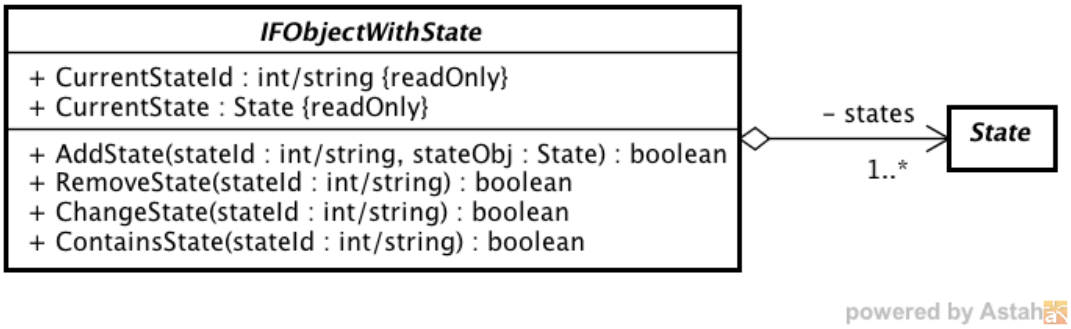


Figura 3.2: La classe IFObjectWithState

La gestione degli stati interni avviene per mezzo dei seguenti metodi:

AddState - aggiunge un oggetto stato alla collezione degli stati interni. Nel caso in cui esista uno stato preesistente con lo stesso identificatore, questo metodo non apporta alcuna modifica alla collezione. L'invocazione di questo metodo al di fuori delle sottoclassi di *IFObjectWithState* è sconsigliato, in quanto solo queste ultime sono in grado di istanziare un oggetto di stato valido (almeno concettualmente).

[1]Ovviamente, una volta scelto che tipo di identificatori usare, non è possibile creare stati con identificatori di tipo differente

RemoveState(stateId) - rimuove dalla collezione degli stati interni lo stato con l'id specificato, se presente. Nel caso in cui si tenti di rimuovere lo stato corrente questo metodo non effettuerà alcuna modifica.

ChangeState(stateId) - Sostituisce lo stato corrente con quello avente l'id specificato, se presente.

Sono stati inoltre definiti i seguenti campi per tenere traccia dello stato corrente:

CurrentStateId - Contiene l'identificatore dello stato corrente.

CurrentState - Contiene il riferimento allo stato corrente. L'accesso a questo campo al di fuori delle sottoclassi di *IFObjectWithState* è sconsigliato.

Ogni sottoclasse di *IFObjectWithState* è libera di limitare la visibilità di questi campi e metodi in base alle proprie necessità. Infine ognuna di queste classi deve creare uno stato di "default" come parte della propria inizializzazione.

3.2 Le classi IFApplication, IFCommand e IFLowLevelEventDescriptor

La classe astratta *IFApplication* rappresenta una applicazione XVR. Essa definisce un insieme di eventi relativi al ciclo di vita dell'applicazione. Fornisce inoltre i meccanismi per la gestione degli stati e delle modalità di interazione.

Abbiamo già introdotto alcune caratteristiche di questa classe nel capitolo 2. In questo paragrafo analizzeremo nel dettaglio gli eventi generati da *IFApplication*, la sua interfaccia ed i servizi che mette a disposizione.

Nota

Nel resto della trattazione non si farà riferimento ai metodi per l'aggiunta e la rimozione di listener. Infatti, dato un evento EVT, le firme di tali metodi seguono sempre la seguente convenzione:

- Il metodo per la registrazione di un listener ha una firma del tipo:
  *void Add*EVT*Listener(listener)*
- Il metodo per la de-registrazione di un listener ha una firma del tipo:
  *void Remove*EVT*Listener(listener)*

Inoltre, quando non indicato diversamente, si deve assumere che il metodo di gestione dell'evento abbia la seguente firma:

- *void On*EVT*()*

Quanto detto si applica sia a questo capitolo che al prosieguo della tesi.

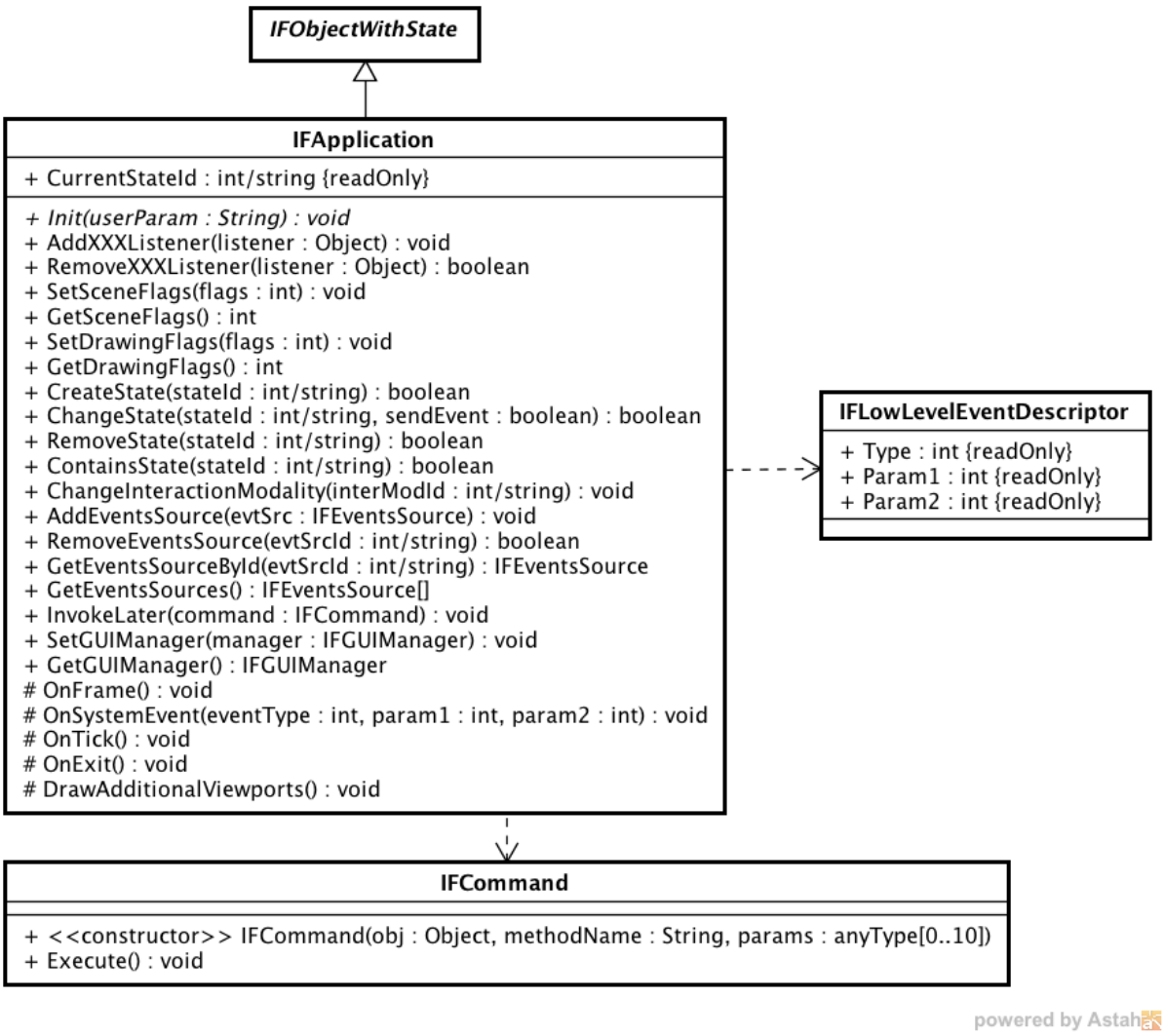


Figura 3.3: Le classi *IFApplication*, *IFCommand* e *IFLowLevelEventDescriptor*

Eventi relativi alle callback di XVR

Come spiegato nel paragrafo 2.1.1, la struttura di una tipico programma S3D consiste in sei *callback*: *OnDownload*, *OnInit*, *OnFrame*, *OnTimer*, *OnEvent* e *OnExit*. La classe *IFApplication* espone (quando possibile) queste *callback* sotto forma di eventi. Nei casi più semplici esiste una corrispondenza biunivoca tra *callback* ed eventi, mentre negli altri casi si è dovuto procedere in maniera differente, come spiegato nella seguente tabella:

Callback di XVR	**Soluzione corrispondente adottata nella classe IFApplication**
OnDownload	Questa funzione è rimasta immutata e deve essere definita esplicitamente. Questo è dovuto al fatto che, al momento della sua invocazione, la "main class" non è stata ancora istanziata.
OnInit	Corrisponde al metodo *Init* di *IFApplication*. Infatti, come spiegato nel paragrafo 2.2.2, *IFApplication* è una classe astratta ed il programmatore è tenuto a definire una sua sottoclasse concreta (la cosiddetta "main class") e a ridefinite il metodo *Init*. Tale metodo dovrà contenere il codice di inizializzazione dell'applicazione.
OnEvent	Corrisponde all'evento *SystemEventFired*. Il metodo di gestione dell'evento deve avere la seguente firma: *void* ***OnSystemEvent****(sysEventDescriptor)* Parleremo del parametro *sysEventDescriptor* nel seguito del paragrafo.
OnTimer	Corrisponde all'evento *Tick*. Il metodo di gestione dell'evento deve avere la seguente firma: *void* ***OnTick****()*
OnExit	Corrisponde all'evento *Exiting*. Il metodo di gestione dell'evento deve avere la seguente firma: *void* ***OnExit****()*

OnFrame	Viene scomposta in tre eventi distinti: • *Rendering*: è l'evento adibito al disegno della scena. I suoi listener sono oggetti disegnabili, ovvero devono implementare il metodo *void* ***Draw****(drawingParams)* Parleremo del parametro *drawingParams* nel seguito del paragrafo. • *PreRendering*: è pensato per l'esecuzione di tutte quelle operazioni che, pur non essendo relative al rendering della scena, devono necessariamente essere eseguite all'interno del blocco *SceneBegin* / *SceneEnd* (ad esempio la funzione *gluUnproject*). Il metodo di gestione dell'evento deve avere la seguente firma: *void* ***OnPreRender****()* • *PostRendering*: viene generato dopo la fase di rendering. In risposta a tale evento vengono eseguite le operazioni periodiche relative alla logica applicativa, all'aggiornamento dello stato dei dispositivi "polling-based" ecc. Il metodo di gestione dell'evento deve avere la seguente firma: *void* ***OnPostRender****()*

Scene flags e Drawing flags

In XVR è possibile influenzare il modo in cui la scena viene renderizzata passando un opportuno flag come parametro della funzione *SceneBegin*. Tali flag[2] sono necessari, ad esempio, per abilitare la modalità di rendering stereoscopico oppure per attivare effetti grafici che si applicano all'intera scena, come il "*motion blur*". È inoltre possibile combinare più flag attraverso l'operatore "bitwise OR". Poiché utilizzando il framework il programmatore non ha più un accesso diretto alla funzione *SceneBegin* (salvo casi particolari che vedremo in seguito), si è reso necessario dotare la classe *IFApplication* di meccanismi per la gestione di questo flag. In particolare, il flag viene ora memorizzato

[2]La lista completa è consultabile alla pagina http://www.vrmedia.it/Docs/A_flagtable.htm

come parte stato corrente dell'applicazione e viene passato come parametro alla funzione *SceneBegin* (in maniera trasparente al programmatore). Vengono inoltri forniti due metodi, **SetSceneFlags** e **GetSceneFlags**, che consentono rispettivamente di settare e di reperire questo parametro.

In maniera analoga, è possibile influenzare il modo in cui vengono disegnati gli elementi della scena passando un flag come parametro del metodo *Draw*. Anche in questo caso vengono forniti due metodi, **SetDrawingFlags** e **GetDrawingFlags** per settare il flag o per ottenerne il valore. È però importante notare che, una volta impostato un "*drawing flag*", questo verrà impiegato per **tutti** gli elementi della scena[3]. In un contesto più generale, questo equivale ad utilizzare tale flag nell'invocazione del metodo *Draw* della radice dello *scene graph*, causandone così la propagazione a tutto il sotto-albero. Qualora si debba disegnare un singolo elemento della scena in un modo diverso da quello predefinito (sfruttando il "*drawing flag*" o modificando lo stato di OpenGL), è necessario ridefinirne il metodo *Draw*. Tuttavia, poiché in S3D non è possibile estendere le classi della libreria standard, questo implica la definizione di una classe che incapsuli l'elemento desiderato e che implementi un metodo *Draw* personalizzato.

Si noti infine che quando questi flag non vengono specificati, i corrispondenti argomenti di *SceneBegin* e *Draw* vengono impostati a *NULL*.

Eventi di sistema

In XVR la gestione degli eventi di sistema viene effettuata in maniera sostanzialmente identica alla gestione dei messaggi di Windows con le API Win32[4]. Infatti, come si può notare, la *callback OnEvent* altro non è che una versione semplificata della "*window procedure*"[5]:

```
// Window procedure
WindowProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)

// OnEvent callback
OnEvent(eventId, param1, param2)
```

I parametri della *OnEvent* rispecchiano quelli della "*window procedure*" (ad eccezione del primo parametro), ed hanno il seguente significato:

eventId - Identifica il tipo di messaggio.

[3] Più precisamente per tutti i listener dell'evento *Rendering*.

[4] Per maggiori dettagli sulla gestione dei messaggi attraverso le API Win32, consultare la MSDN Library alla pagina http://msdn.microsoft.com/en-us/library/windows/desktop/ms632590%28v=vs.85%29.aspx

[5] Funzione responsabile della gestione di tutti i messaggi inviati ad una data finestra.

param1 e **param2** - Contengono informazioni aggiuntive sul messaggio. Il loro contenuto dipende dal tipo di messaggio.

La classe *IFApplication* si limita ad esporre la *OnEvent* sotto forma di evento. In particolare, il metodo per la gestione dell'evento *SystemEventFired* deve avere la seguente firma:

void OnSystemEvent(sysEventDescriptor)

dove il parametro *sysEventDescriptor* è un'istanza della classe *IFLowLevelEventDescriptor,* mostrata in figura 3.3. Come si può notare, questa classe è semplicemente una struttura contenente i tre parametri di cui sopra.

Ovviamente questo approccio non semplifica minimamente la gestione degli eventi rispetto alla *OnEvent*, ed in effetti non è questo il suo fine: la scelta di definire l'evento *sysEventDescriptor* è stata fatta solo per ragioni di uniformità. Dobbiamo però tener conto del fatto che tale evento è destinato principalmente alle classi che modellano dispositivi di input, e solo in circostanze particolari deve essere gestito esplicitamente dal programmatore. Infatti, sono proprio le classi che modellano dispositivi di input a semplificare la gestione dell'input, in quanto il loro scopo è quello di filtrare i soli messaggi significativi, decodificarli e generare in risposta ad essi eventi di alto livello. Queste classi verranno trattate nel capitolo 4.

La tabella delle fonti di eventi

Come spigato nel capitolo 2, al fine di dare la possibilità la possibilità ad un listener di reperire un riferimento alla propria fonte di eventi senza creare un ciclo di riferimenti (con conseguente *memory leak*), tutte le fonti di eventi[6] vengono inserite in una apposita tabella hash gestita da *IFApplication.* I metodi per la gestione di tale tabella sono i seguenti:

AddEventsSource - Aggiunge una fonte di eventi alla tabella.

RemoveEventsSource - Rimuove una fonte di eventi dalla tabella.

GetEventsSourceById - Restituisce un riferimento alla fonte di eventi associata all'identificatore passato come parametro.

GetEventsSources - Restituisce un array contenente i riferimenti a tutte le fonti di eventi presenti in tabella.

[6] Istanze di una sottoclasse concreta *IFEventsSource.*

Tutte le classi che definiscono degli eventi devono estendere *IFEventsSource*, descritta nel paragrafo 3.3. L'unica eccezione è rappresentata proprio da *IFApplication.* Questo perché *IFEventsSource* prevede che ogni fonte di eventi sia associata ad un identificatore e che fornisca un metodo *Register* per l'inserimento in tabella. Come è evidente nessuna delle due condizioni è applicabile a *IFApplication.*

Stati e modalità di interazione

Nel capitolo 2 sono stati introdotti i concetti di **stato dell'applicazione** e **modalità di interazione**. Vediamo ora nel dettaglio gli eventi ed i metodi che la classe *IFApplication* mette a disposizione per supportare queste funzionalità. Partiamo dagli stati ed in particolare dalla loro composizione. Per **stato dell'applicazione**, o più precisamente stato della classe *IFApplication*, si intende un insieme costituito dai seguenti oggetti:

1. I listener di tutti gli eventi definiti da *IFApplication* e tutti gli oggetti da essi raggiungibili.
2. Le fonti di eventi e tutti gli oggetti da esse raggiungibili.
3. Un GUI Manager ed un insieme di controlli grafici.
4. *Scene flag* e *Drawing flag.*

I metodi messi a disposizione per la gestione degli stati sono:

CreateState - Crea un nuovo stato con l'id specificato, senza però modificare lo stato corrente.

RemoveState - Elimina uno stato. L'operazione fallisce se si cerca di eliminare lo stato corrente.

ContainsState - Verifica l'esistenza di uno stato a partire dal suo identificatore.

ChangeState - Esegue un cambiamento di stato.

Prima che un cambiamento di stato venga effettivamente portato a termine, viene generato l'evento *StateWillChange.* Tale evento ha lo scopo di consentire agli oggetti interessati (tipicamente controller) di riportare il proprio stato interno in una condizione di consistenza prima del cambio di stato dell'applicazione. La generazione dell'evento *StateWillChange* può però risultare dannosa in fase di inizializzazione (ovvero nel metodo *Init*). Per tale ragione il metodo *ChangeState* accetta un parametro per specificare se generare o meno tale evento.

Si noti che è possibile ottenere l'identificatore dello stato corrente attraverso il campo *CurrentStateId* ereditato da *IFObjectWithState*. Tuttavia i rimanenti campi e metodi di *IFObjectWithState* non devono essere usati direttamente.

Infine è bene ricordare che esiste uno stato di "*default*", il cui identificatore deve essere esplicitamente definito all'inizio di ogni programma attraverso la macro IFC_APP_-DEFAULT_STATE_ID, come spigato nel paragrafo 2.2.3.

Per quanto riguarda le modalità di interazione, dal punto di vista implementativo altro non sono che stati interni degli oggetti controller. Per tale ragione *IFApplication* non si occupa della loro gestione, ma si limita a fornire un meccanismo di notifica per mezzo del quale richiedere ai controller il cambio di modalità. Il metodo preposto a tale scopo è **ChangeInteractionModality**, che prende come parametro l'identificatore della nuova modalità. Questo metodo si limita sostanzialmente a generare due eventi: *InteractionModalityWillChange* e *InteractionModalityChange*. Il primo ha lo scopo di consentire agli oggetti interessati di riportare il proprio stato interno in una condizione di consistenza prima del cambio di modalità. Il secondo rappresenta invece la richiesta vera e propria, ed è destinato ai soli oggetti controller. Infatti il metodo di gestione dell'evento è proprio *ChangeInteractionModality* della classe *IFController* (torneremo su questo punto nel paragrafo 3.4).

GUI

Come vedremo nel capitolo 7, l'interfaccia grafica viene gestita da un'istanza della classe *IFGUIManager*. Una volta creata l'interfaccia grafica, il processo di aggiornamento e rendering della stessa è completamente trasparente al programmatore. Per far ciò, però, il GUI manager necessita della collaborazione della classe *IFApplication*. Il manager viene memorizzato nello stato dell'applicazione, in modo da avere un'interfaccia grafica diversa in ogni stato. I metodi per aggiungere il manager allo stato corrente e per reperirne il riferimento sono rispettivamente **SetGUIManager** e **IFGUIManager GetGUIManager**. Infine, per eliminare un manager dallo stato corrente basta invocare *SetGUIManager* passando *NULL* coma parametro.

InvokeLater e IFCommand

Nel paragrafo 2.2.6 abbiamo visto il meccanismo di l'esecuzione differita, che consente di eseguire uno o più comandi al termine di ogni frame, dopo che è terminata la fase di gestione degli eventi. L'idea chiave è quella di incapsulare questi comandi in un oggetto, detto "*command*". In particolare, ogni oggetto "*command*" deve implementare il metodo

void Execute()

che si occupa dell'esecuzione dei comandi incapsulati. Una volta creati, questi oggetti devono essere inseriti in una apposita coda di esecuzione gestita da *IFApplication*, per mezzo del metodo **InvokeLater**.

Poiché spesso tali comandi consistono nell'invocazione di un singolo metodo su di un oggetto, e per evitare di dover definire una classe solo per far questo, è stata definita nel framework una apposita classe di utilità: *IFCommand*. Come si può notare in figura 3.3, la classe *IFCommand* definisce soltanto un costruttore ed il metodo *Execute*. I parametri del costruttore sono i seguenti:

object - il riferimento all'oggetto sul quale si desidera invocare un metodo.

methodName - una stringa contenente il nome del metodo da invocare.

param1 ... param10 - i parametri da passare al metodo (opzionali).

Estendere IFApplication

Come spiegato nei paragrafi 2.2.2 e 2.2.3, lo sviluppo ogni applicazione che utilizza il framework di interazione parte dalla definizione di una "main class" che estende *IFApplication*. Segue che è possibile adattare il comportamento di *IFApplication* in base specifiche necessità semplicemente sovrascrivendone alcuni metodi, oppure definendo campi e metodi aggiuntivi. Tipicamente ad essere ridefiniti sono i metodi corrispondenti alle callback predefinite di XVR, per illustrare i quali occorre aprire una piccola parentesi. Il framework comprende un "main file", nascosto al programmatore, nel quale viene istanziata la "main class" e vengono definite le *callback* di XVR. La struttura di questo file è molto simile a quella del template visto nel paragrafo 2.1.1 a pagina 33. In particolare, l'implementazione delle *callback* consiste nell'invocazione di un metodo speculare definito dalla classe *IFApplication*: **OnFrame**, **OnSystemEvent**, **OnTick** e **OnExit**. In genere la loro ridefinizione è abbastanza semplice: basta invocare alla fine del metodo sovrascritto l'implementazione della superclasse. Ad esempio:

```
// Corrisponde alla funzione OnEvent di XVR
function IFAPPNAME::OnSystemEvent(eventType, param1, param2)
{
   // Codice necessario alla nostra applicazione
   [...]

   // Invoca IFApplication::OnSystemEvent()
   IFApplication::this.OnSystemEvent(eventType, param1, param2);
}
```

L'unica eccezione, per la quale è previsto un procedimento particolare, è *OnFrame*. Questo è anche il metodo che più spesso viene sovrascritto, ad esempio ogni volta che si ha bisogno di più blocchi *SceneBegin / SceneEnd* al fine di disegnare su *viewport* multipli oppure per produrre immagini stereoscopiche. Per venire in contro a tali necessità, *IFApplication* dichiara il metodo **DrawAdditionalViewports**[7], la cui implementazione normalmente è vuota, ma che può essere sovrascritto dalla "main class" per inserirvi blocchi *SceneBegin / SceneEnd* aggiuntivi[8]. Supponiamo ad esempio di aver bisogno di un secondo *viewport* posizionato nell'angolo in alto a destra della finestra. Il metodo *DrawAdditionalViewports* può essere ridefinito come segue:

```
function IFAPPNAME::DrawAdditionalViewports()
{
   // Disegna il secondo viewport.
   // I parametri della SceneBeginRel
   // ne definiscono posizione e dimensione.
   SceneBeginRel(0.8,0.7,0.2,0.3);
   [...]
   SceneEnd();
}
```

Qualora anche questo non sia sufficiente, è sempre possibile ridefinire la *OnFrame* partendo dall'implementazione data in *IFApplication*.

3.3 La classe IFEventsSource

La classe astratta *IFEventsSource* rappresenta una generica fonte di eventi di alto livello. Esempi tipici sono le classi che modellano un dispositivo di input, oppure le gli oggetti del modello dell'applicazione che sfruttano il meccanismo degli eventi per inviare notifiche riguardo ad un cambiamento del loro stato interno.

[7] Questo metodo viene invocato dopo la generazione dell'evento *Rendering*, ma prima di disegnare la GUI e di generare l'evento *PostRendering*.

[8] Concettualmente, è stato applicato il pattern *Template Method* [16].

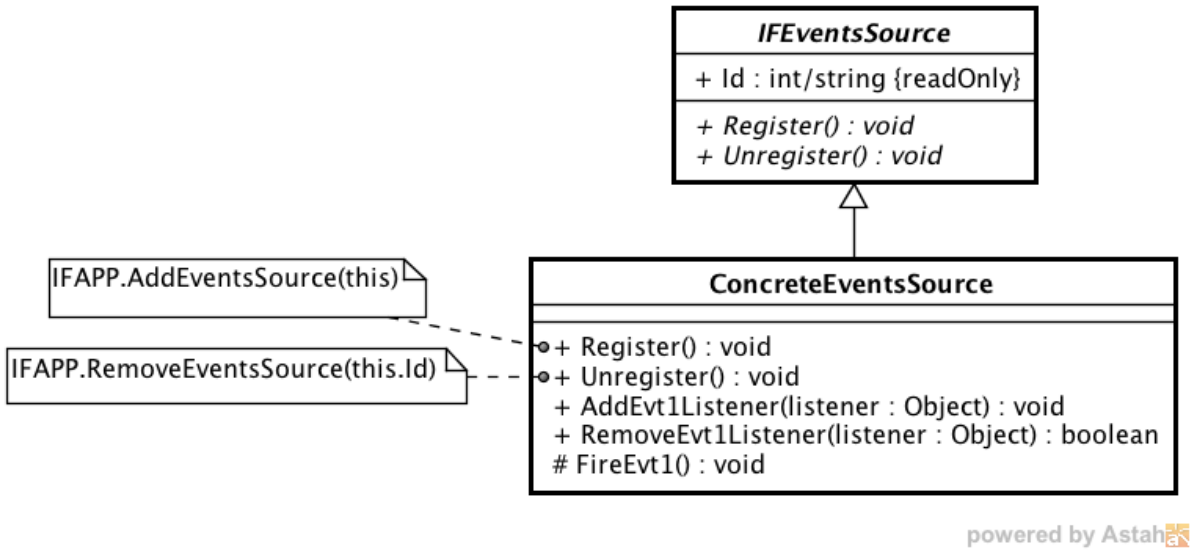


Figura 3.4: La classe *IFEventsSource*

Questa classe non impone alcun vincolo sul tipo di eventi generati dalle sue sottoclassi concrete, né sui metodi che queste mettono a disposizione per la gestione dei listener. Si suppone infatti che nel definire una sorgente di eventi, il programmatore rispetti le convenzioni descritte nel paragrafo 2.2.5. Le uniche regole che questa classe impone, e che il programmatore deve rispettare nell'implementazione di una fonte di eventi, sono relative alla tabella delle fonti di eventi della classe *IFApplication*. In particolare:

- Ad ogni fonte di eventi è associato un identificatore unico, di tipo intero o una stringa. Tale identificatore deve essere memorizzato nel campo pubblico ***Id***.
- Ogni sorgente deve implementare il metodo
  void Register()
  che registra la sorgente stessa nella tabella delle fonti di eventi della classe *IFApplication*. Se l'oggetto ha bisogno di ricevere eventi da altre sorgenti (come accade per gli oggetti che rappresentano dei dispositivi di input), questo metodo si occupa anche della registrazione presso tali sorgenti.
- Ogni sorgente deve implementare il metodo
  void Unregister()
  che effettua la de-registrazione dalla tabella delle fonti di eventi e da eventuali altre fonti di eventi da cui l'oggetto dipende.

Infine, poiché è molto comune registrare un oggetto subito dopo averlo creato, si consiglia di implementare il costruttore in modo tale invocare automaticamente la procedura di registrazione subito dopo l'inizializzazione dell'oggetto stesso, come spiegato nel paragrafo 2.2.5.

3.4 La classe IFController

Lo scopo degli oggetti "*controller*" è quello di modificare il modello dell'applicazione in risposta ad eventi, siano essi eventi di input o notifiche di altra natura. La classe astratta IFCommand dichiara i metodi che devono essere implementati da tutti i controller. Tali metodi riguardano la gestione delle *modalità di interazione*, nonché le modalità con cui i controller si registrano presso le sorgenti di eventi.

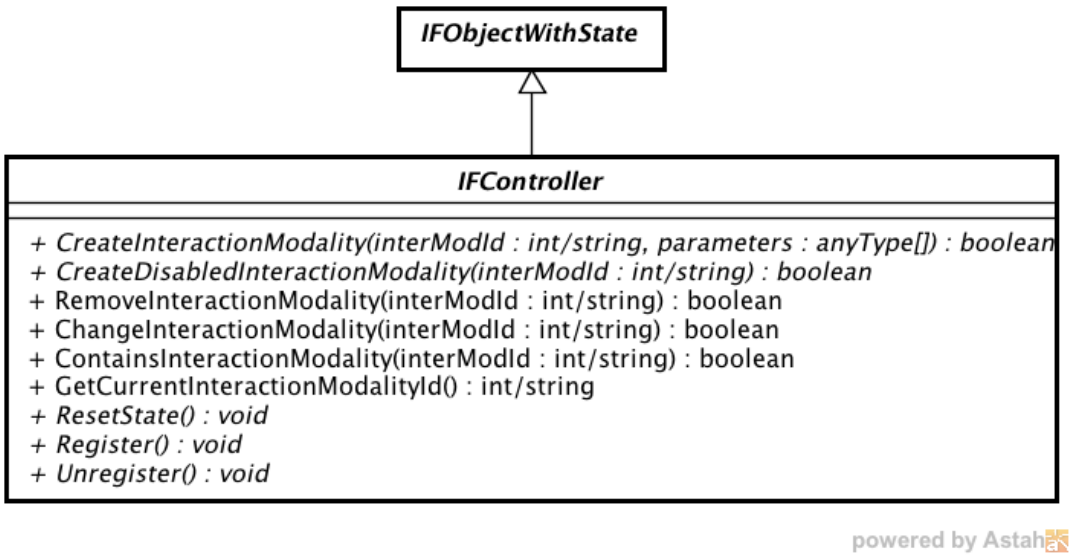


Figura 3.5: La classe *IFController*

Una *modalità di interazione* rappresenta una particolare configurazione dello stato interno di un oggetto controller e condiziona il suo comportamento in risposta agli eventi. In altri termini, al variare della modalità di interazione ogni controller può intraprendere azioni differenti in risposta allo stesso tipo di eventi. Dal punto di vista dell'utente, modalità di interazione differenti corrispondono ad un diverso modo di interagire all'interno dell'ambiente virtuale, dovuto all'uso metafore o tecniche di interazione differenti.

Seguendo la definizione data viene naturale implementare le modalità di interazione sfruttando il pattern *State* [16] e di conseguenza le funzionalità offerte dalla classe *IFObjectWithState*, da cui *IFController* deriva[9].

I metodi messi a disposizione da *IFController* per la gestione delle modalità sono:

CreateInteractionModality - Crea una nuova modalità di interazione, senza modificare la modalità corrente. Prende come argomenti l'identificatore da associare alla nuova modalità e un array contenente i parametri relativi al funzionamento della

[9]Tuttavia, a meno che non si stia definendo un nuovo controller, i campi ed i metodi di *IFObjectWithState* non non sono accessibili direttamente.

modalità stessa. Il numero e la tipologia di tali parametri dipende dallo specifico controller. È un metodo astratto, che deve essere opportunamente definito da ogni sottoclasse di *IFController*.

CreateDisabledInteractionModality - Crea una modalità di interazione "disabilitata", ovvero una modalità di interazione in cui il controller non esegue alcuna azione in risposta agli eventi che gli vengono notificati (semplicemente li ignora).

RemoveInteractionModality - Elimina la modalità di interazione specificata (purché esista e non sia la modalità corrente).

ChangeInteractionModality - Modifica lo stato del controller passando alla modalità di interazione specificata. Si noti che se la modalità richiesta non esiste, il controller mantiene lo stato corrente. Questo è anche il metodo chiamato a gestire l'evento *InteractionModalityChange* della classe *IFApplication* (in effetti è abbastanza raro che venga invocato in situazioni differenti).

ContainsInteractionModality - Verifica l'esistenza di una modalità a partire dal suo identificatore.

GetCurrentInteractionModalityId - Restituisce l'identificatore associato alla modalità corrente.

È importante cogliere la differenza tra l'assenza di una modalità e la sua esistenza sotto forma di modalità "disabilitata". Supponiamo infatti che venga richiesto ad un controller di passare alla modalità "MOD5" (per mezzo del metodo *ChangeInteractionModality*). Se il controller possiede una modalità "disabilitata" associata all'identificatore "MOD5", questa diverrà la modalità corrente e da quel momento in poi ogni evento ricevuto verrà ignorato. Se invece non esiste alcuna modalità con quell'identificatore, il controller semplicemente ignorerà la richiesta e continuerà a gestire gli eventi secondo la modalità precedente. Questo tipo di funzionamento risulta utile per creare controller che supportano solo un sottoinsieme delle modalità previste nell'applicazione, in alternativa al definire più modalità con lo stesso identico comportamento[10]. Quest'ultimo approccio (sconsigliato a meno di una attenta valutazione) nasconde inoltre un problema subdolo. Supponiamo infatti di creare due modalità con il medesimo comportamento e con l'idea che producano gli stessi effetti di una modalità unica. Poiché le modalità sono di fatto oggetti che incapsulano una parte dello stato del controller, quando si passa dall'una all'altra quella parte di stato cambierà, producendo bug di difficile individuazione.

Di particolare importanza è il metodo **ResetState**. Il suo scopo è quello di "resettare" lo stato interno del controller con riferimento alla modalità corrente. Viene usato principalmente in risposta agli eventi *StateWillChange* e *InteractionModalityWillChange* della

[10] Ottenuto passando il medesimo array di parametri al metodo *CreateInteractionModality*.

classe *IFApplication*, al fine di riportare lo stato interno del controller in una condizione di consistenza prima di un cambio di stato o di modalità.

Infine la classe IFCommand dichiara gli ormai classici metodi **Register** e **Unregister** che si occupano di (de-)registrare il controller presso le fonti di eventi.

Vediamo ora come creare un nuovo controller. I passi da seguire per definire una sottoclasse di *IFController* sono:

- Definire una classe che rappresenti una generica modalità di interazione ed eventualmente un'altra per le modalità "disabilitate".
- Ricordarsi di creare nel costruttore una modalità di "default" avente IFC_CTRL_DEFAULT_INTERMOD_ID come identificatore.
- Implementare i metodi astratti *CreateInteractionModality*, *CreateDisabledInteractionModality*, *ResetState*, *Register* e *Unregister*.

Per chiarire i concetti spiegati, proviamo ad implementare un semplice controller che conta il numero di volte che l'utente preme un carattere sulla tastiera e lo stampa a schermo. Supponiamo che ogni modalità si riferisca ad un diverso carattere da conteggiare. Il codice del controller di esempio è riportato di seguito:

```
// Classe che implementa le modalità di interazione.
// Rappresenta una parte dello stato interno di CharCounter.
class CharCounterInterMod
{
   OnKeyUp(keyDescriptor);
   ResetState();

   var characterCode; // codice ASCII del carattere da conteggiare
   var counter; // contatore
};

// Costruttore
function CharCounterInterMod::CharCounterInterMod(aCharacterCode)
{
   characterCode = aCharacterCode;
   counter = 0;
}

// Al rilascio di un tasto della tastiera...
function CharCounterInterMod::OnKeyUp(keyDescriptor)
{
   if(keyDescriptor.KeyCode == characterCode)
   {
```

```
      counter++;
      OutputIn(char(characterCode), ":␣", counter);
   }
}

// Resettare la modalità consiste
// nell'azzerare il contatore (in questo esempio)
function CharCounterInterMod::ResetState() { counter = 0; }

// La classe che implementa il "controller"
class CharCounter:IFController
{
   // Metodi astratti di IFController (da definire)
   CreateInteractionModality(interModId, parameters);
   CreateDisabledInteractionModality(interModId);
   Register();
   Unregister();
   ResetState();

   // Gestori degli eventi
   OnKeyUp(keyDescriptor);
   OnInteractionModalityWillChange(interModId);
   OnStateWillChange(stateId);

   // Contiene l'Id dell'oggetto "tastiera".
   // Servirà per reperire il riferimento alla
   // tastiera nei metodi Register e Unregister.
   var keyboardId;
};

// Costruttore
function CharCounter::CharCounter(keyboard, autoregister, aCharacter)
{
   keyboardId = keyboard.Id; // Si memorizza l'Id dell'oggetto "tastiera"

   // Si crea la modalità di "default".
   // Il parametro aCharacter rappresenta il carattere di cui si vogliono
   // contare le pressioni in questa modalità.
   // È opzionale. Se viene omesso aCharacter verrà settato a NULL.
   // In tal caso la modalità di default sarà "disabilitata".
   if(aCharacter != NULL)
      CreateInteractionModality(IFC_CTRL_DEFAULT_INTERMOD_ID, {aCharacter});
   else
      CreateDisabledInteractionModality(IFC_CTRL_DEFAULT_INTERMOD_ID);

   // Se autoregister viene impostato a true si effettua la registrazione
   // al termine dell'inizializzazione dell'oggetto.
```

```
    if(autoregister) Register();
}

// Crea una nuova modalità di interazione
function CharCounter::CreateInteractionModality(interModId, parameters)
{
    // Crea una modalità di interazione passando come parametro
    // il codice ASCII del carattere da conteggiare.
    var intermod = CharCounterInterMod(parameters[0]);

    // Una volta creato, l'oggetto "modalità" viene memorizzato
    // sfruttando il metodo "AddState" di IFObjectWithState
    return AddState(interModId, intermod);
}

// Crea una modalità di interazione "disabilitata".
function CharCounter::CreateDisabledInteractionModality(interModId)
{
    // In generale si sarebbe dovuto definire una classe
    // apposita ma, vista la semplicità del'esempio,
    // basta creare una modalità standard passando come
    // parametro il carattere NULL
    CreateInteractionModality(interModId, {0})
}

// Resetta la modalità corrente.
function CharCounter::ResetState()
{
    // Si noti che il riferimento alla modalità corrente
    // è memorizzato nel campo "CurrentState" di IFObjectWithState
    CurrentState.ResetState();
}

// Registra il controller presso le fonti di eventi
function CharCounter::Register()
{
    // Si usa il metodo "GetEventsSourceById" di IFAPP
    // per ottenere un riferimento all'oggetto tastiera.
    var keyboard = IFAPP.GetEventsSourceById(keyboardId);

    // Registrazione
    keyboard.AddKeyUpListener(this);
    IFAPP.AddInteractionModalityChangeListener(this);
    IFAPP.AddStateWillChangeListener(this);
    IFAPP.AddInteractionModalityWillChangeListener(this);
}
```

```
// De-registra il controller
function CharCounter::Unregister()
{
   var keyboard = IFAPP.GetEventsSourceById(keyboardId);

   keyboard.RemoveKeyUpListener(this);
   IFAPP.RemoveInteractionModalityChangeListener(this);
   IFAPP.RemoveStateWillChangeListener(this);
   IFAPP.RemoveInteractionModalityWillChangeListener(this);
}

// Gestisce l'evento KeyUp
function CharCounter::OnKeyUp(keyDescriptor)
{
   // In accordo al pattern State, la gestione
   // viene delegata allo stato (modalità) corrente.
   CurrentState.OnKeyUp(keyDescriptor);
}

// Gestisce l'evento InteractionModalityWillChange
function CharCounter::OnInteractionModalityWillChange(interModId)
{
   // Prima di un cambio di modalità
   // si resetta la modalità corrente.
   if(CurrentStateId != interModId) ResetState();
}

// Gestisce l'evento StateWillChange
function CharCounter::OnStateWillChange(stateId)
{
   // Prima di un cambio di stato
   // si resetta la modalità corrente.
   ResetState();
}
```

L'utilizzo di un controller così definito è semplicissimo: basta istanziarlo ricordandosi di impostare a true il parametro "autoregister":

```
IFKeyboard keyboard = ...;
var counter = CharCounter(keyboard, true, VK_SPACE);
```

Una volta creato il controller mi metterà automaticamente in ascolto degli eventi da tastiera e non necessita quindi di ulteriori interventi da parte del programmatore (a meno che non si vogliano definire altre modalità di interazione).

Capitolo 4

Il package “Devices”

In questo package sono definite le classi che modellano i dispositivi di input supportati dal framework. In questo contesto, per dispositivo si intende una generica fonte di eventi di input.

4.1 La classe IFDevice

La classe astratta *IFDevice* modella una generica fonte di eventi di input, tipicamente un dispositivo fisico. Lo scopo principale di un dispositivo (sottoclasse di *IFDevice*) è quello di generare eventi di alto livello in corrispondenza della variazione del suo stato interno. L’aggiornamento dello stato interno può avvenire in vari modi, a seconda della natura del dispositivo. Tipicamente l’aggiornamento avviene in conseguenza della gestione di un evento *SystemEventFired* generato dall’oggetto “applicazione”. In questo caso l’oggetto dispositivo si deve far carico di filtrare i soli messaggi significativi (cioè quelli che gli competono) e di decodificarne i parametri. È anche possibile che la fonte di eventi di input sia in realtà un’applicazione remota, che comunica con l’applicazione XVR inviando messaggi su protocollo UDP o TCP. Infine, esistono casi in cui lo stato del dispositivo reale debba essere interrogato periodicamente[1] per mezzo di una apposita API. Si parla allora di dispositivi *polling-based*.

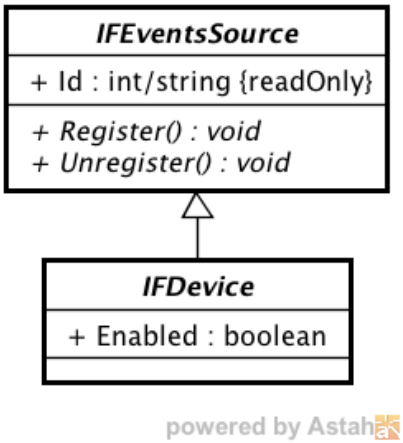


Figura 4.1: La classe *IFDevice*

Come si vede in figura 4.1, la classe *IFDevice* si limita ad estendere *IFEventsSource* definendo il campo booleano **Enabled**, che consente di abilitare o disabilitare il dispositivo.

[1] Tipicamente alla ricezione dell’evento *PostRendering* della classe *IFApplication*.

Se un dispositivo viene disabilitato, esso non genererà più eventi, pur continuando ad aggiornare il suo stato interno.

Nel definire una sottoclasse di *IFDevice* è importante ricordarsi di accettare un identificatore come parametro del costruttore e di definire i metodi **Register** e **Unregister**, attraverso i quali il dispositivo si (de)registra presso le fonti di eventi di cui necessita (tipicamente presso l'oggetto "applicazione" e relativamente agli eventi *SystemEventFired* e *PostRendering*).

4.2 La classe IFKeyboard

Il primo dispositivo che esamineremo è la tastiera, modellata dalla classe *IFKeyboard*. Essa prevede solo due tipi di eventi, **KeyDown** e **KeyUp**, generati rispettivamente quando l'utente preme e rilascia un tasto. I relativi listener devono implementare rispettivamente i metodi:

void OnKeyUp(descriptor)

e

void OnKeyDown(descriptor)

dove il parametro "*descriptor*" è un'istanza della classe *IFKeyDescriptor*, mostrata in figura 4.2. Il descrittore contiene:

- Il codice associato al tasto premuto/rilasciato. Se il tasto corrisponde ad un carattere questo campo conterrà il suo codice ASCII, altrimenti si ricorre a speciali costanti dette "*virtual-key codes*"[2].
- Lo stato dei tasti "modificatori" (ALT, CONTROL e SHIFT), espresso come la combinazione (in *bitwise-or*) delle costanti IFC_KEY_SHIFT_PRESSED, IFC_KEY_CONTROL_PRESSED e IFC_KEY_ALT_PRESSED. Se nessuno di questi tasti è premuto, viene usata la costante IFC_KEY_NO_MODIFIERS.
- Un flag booleano che indica se l'evento è stato generato a seguito di una reale pressione del tasto o in conseguenza della funzionalità di *auto-repeat*. Questo campo è significativo solo per l'evento *KeyDown*.

[2]L'elenco completo è riportato alla pagina http://www.vrmedia.it/Docs/B_vkeytable.htm

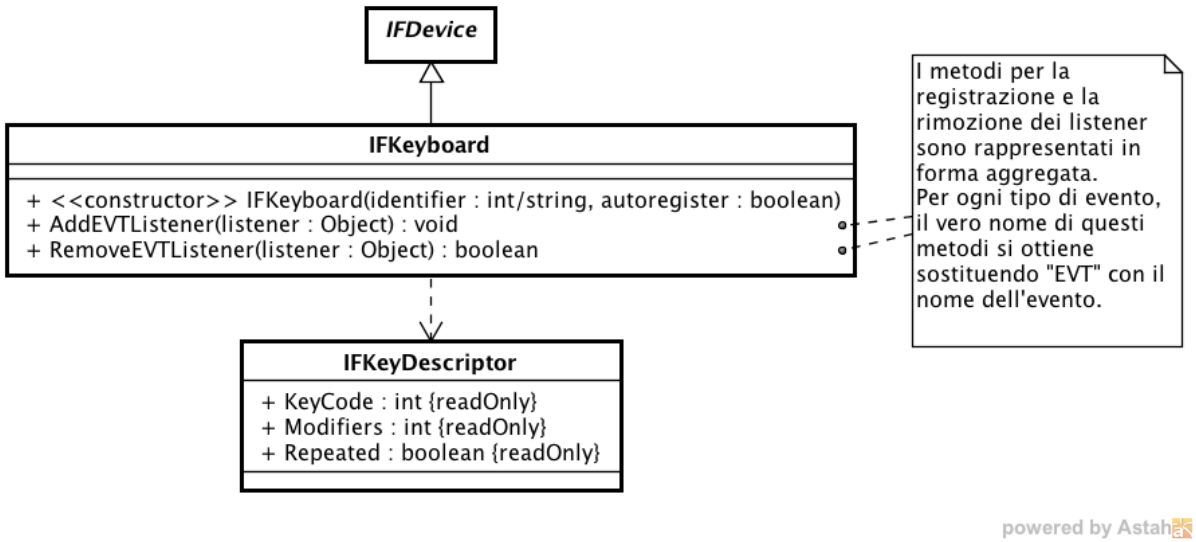


Figura 4.2: Le classi *IFKeyboard* e *IFKeyDescriptor*

I metodi per la registrazione e la rimozione dei listener vengono indicati in questo capitolo rispettivamente come **AddEvtListener** e **RemoveEvtListener**, dove Evt rappresenta il nome di uno qualsiasi degli eventi generati dal dispositivo in oggetto (*KeyUp* e *KeyDown* in questo caso).

Per quanto riguarda il costruttore, esso accetta due argomenti, che rappresentano rispettivamente l'identificatore (di tipo intero o stringa) associato al dispositivo e un flag booleano per richiederne la registrazione al termine della sua inizializzazione. Quasi tutte le classi che estendono *IFDevice* hanno un costruttore che accetta gli stessi argomenti.

4.3 La classe IFMouse

Il mouse viene modellato dalla classe *IFMouse*. Questa classe si occupa della decodifica dei messaggi di basso livello del mouse e della generazione di un insieme di eventi di alto livello ad essi correlati, compresi quelli relativi ai concetti di "*click*"[3] e "*drag*"[4].

[3]Pressione e rilascio di un pulsante del mouse senza che venga spostato il cursore tra i due eventi.
[4]Spostamento del cursore mantenendo premuto un pulsante.

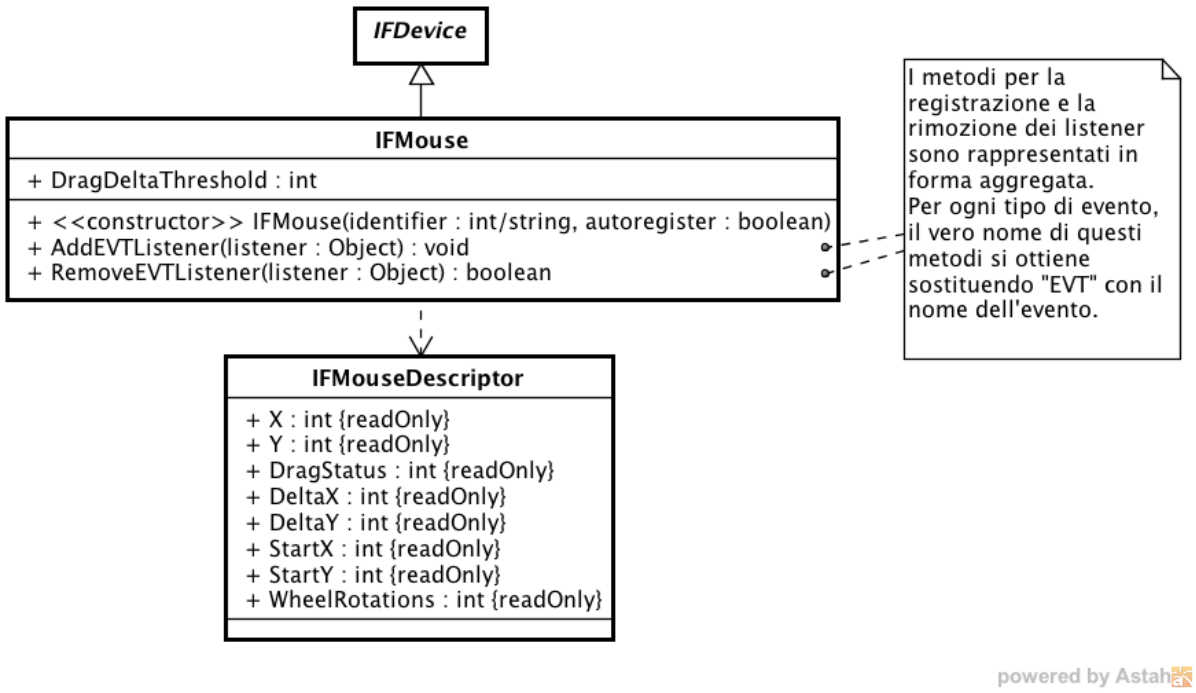


Figura 4.3: Le classi *IFMouse* e *IFMouseDescriptor*

Nel dettaglio, gli eventi generati da *IFMouse* sono:

LeftDown, **RightDown**, **MiddleDown** - Vengono generati quando l'utente preme rispettivamente il pulsante sinistro, destro o centrale del mouse.

LeftUp, **RightUp**, **MiddleUp** - Vengono generati quando l'utente rilascia rispettivamente il pulsante sinistro, destro o centrale del mouse.

LeftClick, **RightClick**, **MiddleClick** - Vengono generati quando l'utente preme e rilascia rispettivamente il pulsante sinistro, destro o centrale del mouse senza spostare il cursore tra la pressione ed il rilascio (cioè senza effettuare un "*drag*").

LeftDoubleClick, **RightDoubleClick**, **MiddleDoubleClick** - Vengono generati quando l'utente preme e rilascia due volte ed in rapida successione rispettivamente il pulsante sinistro, destro o centrale del mouse.

Move - Generato ogni volta che l'utente sposta il cursore del mouse (anche in caso di *drag*).

StartDrag - Generato all'inizio di una operazione di *drag*.

Drag - Generato ogni volta che l'utente sposta il cursore del mouse tenendo premuto un pulsante.

EndDrag - Generato alla fine di una operazione di *drag*.

Wheel - Generato ogni volta che l'utente agisce sulla "rotella" del mouse.

Si noti che l'operazione di *drag* viene riconosciuta tale solo se l'utente sposta il cursore di una distanza superiore ad una certa soglia, configurabile per mezzo del campo **DragDeltaThreshold**.

I metodi implementati dai listener per la gestione di questi eventi hanno tutti una firma del tipo:

void OnEvt(descriptor)

dove Evt rappresenta il nome degli eventi di cui sopra ed il parametro "*descriptor*" è un'istanza della classe *IFMouseDescriptor*. Il descrittore fornisce informazioni aggiuntive sull'evento e sullo stato del mouse al verificarsi dello stesso. In particolare esso è costituito dai seguenti campi (accessibili in sola lettura):

X e **Y** - Contengono le coordinate correnti del cursore.

DeltaX e **DeltaY** - Il loro valore è significativo solo per gli eventi di tipo *Move* e *Drag*. Contengono l'entità dell'ultimo spostamento del cursore.

DragStatus - Consente di sapere se è in corso un *drag* e quale pulsante è coinvolto. Questo campo può assumere uno dei valori definiti dalle seguenti costanti: IFC_MOUSE_NO_DRAG, IFC_MOUSE_LEFT_BUTTON, IFC_MOUSE_RIGHT_BUTTON e IFC_MOUSE_MIDDLE_BUTTON.

StartX e StartY - Il loro valore è significativo solo per gli eventi di tipo *StartDrag*, *Drag* e *EndDrag*. Contengono le coordinate del punto in cui è iniziato il *drag*.

WheelRotations - Contiene il numero di "scatti" percorsi dalla "rotella" del mouse durante l'ultima rotazione. Questo campo è significativo solo per eventi di tipo *Wheel*.

Si noti che le costanti IFC_MOUSE_LEFT_BUTTON, IFC_MOUSE_RIGHT_BUTTON e IFC_MOUSE_MIDDLE_BUTTON vengono usate, oltre che per il campo DragStatus del descrittore, anche come parametri dei controller del mouse, al fine di associare ai vari pulsanti le funzionalità messe a disposizione dal controller stesso.

4.4 La classe IFTouchscreen

Il framework supporta i touchscreen a "tocco singolo" grazie alla classe *IFTouchscreen*. Gli eventi generati da questo dispositivo sono:

TouchDown / **TouchUp** - Generati rispettivamente quando l'utente inizia e termina un "tocco".

Tap - Generato quando l'utente tocca e rilascia lo schermo senza spostare il dito.

StartDrag - Generato all'inizio di una operazione di *drag*.

Drag - Generato ogni volta che l'utente sposta il dito sullo schermo.

EndDrag - Generato alla fine di una operazione di *drag*.

Il descrittore degli eventi è un'istanza della classe *IFTouchscreenDescriptor*, mostrata in figura 4.4. I campi in esso contenuti sono un sottoinsieme di quelli definiti da *IFMouseDescriptor*.

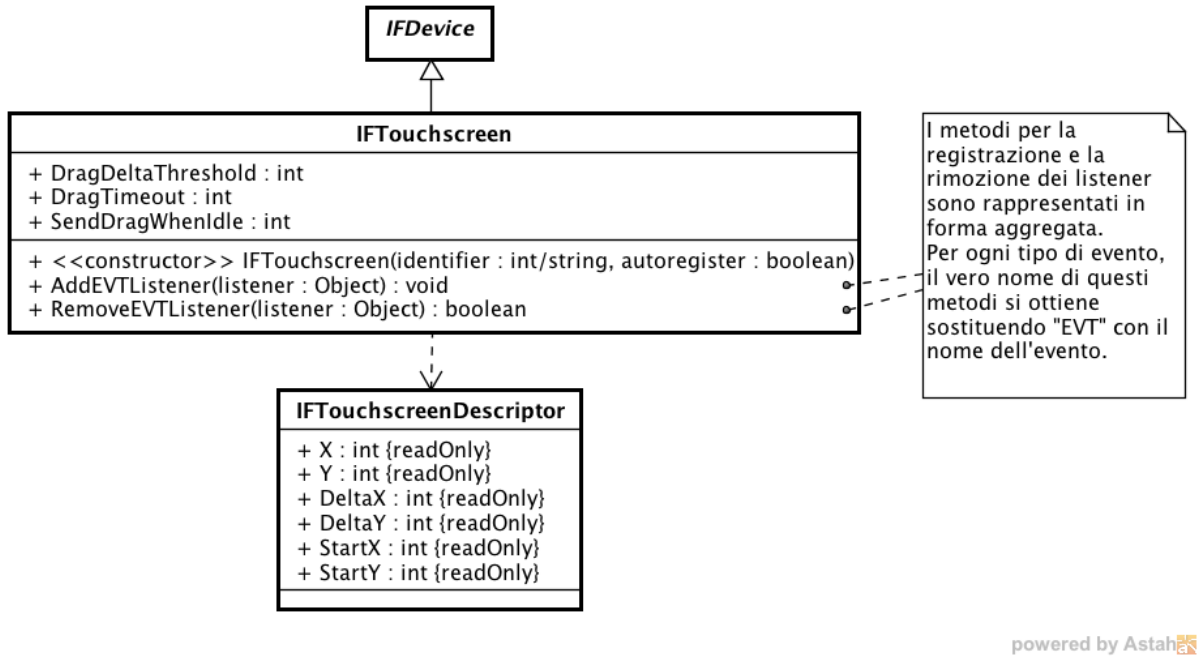


Figura 4.4: Le classi *IFTouchscreen* e *IFTouchscreenDescriptor*

L'implementazione della classe *IFTouchscreen* si basa sulla gestione degli stessi messaggi usati per il mouse. Tuttavia per questo dispositivo sono necessarie alcune accortezze particolari, dovute alla non perfetta rilevazione del tocco da parte di molti dispositivi fisici. È infatti comune, provando a tracciare una linea curva sullo schermo, che questa si interrompa per poi riprendere poco dopo, come mostrato in figura 4.5 (a). Questo effetto è causato dalla scarsa sensibilità di alcune regioni dello schermo e/o da una pressione non uniforme del tocco. Per risolvere questo problema, la classe *IFTouchscreen* ricorre ad un timer che si attiva ogni volta che viene rilevata un'interruzione del tocco. Se prima del "timeout" il tocco riprende, allora si è trattato di uno dei casi sopra descritti. La generazione dell'evento *TouchUp* viene quindi annullata e viene generato un evento

Drag con un "delta" tale da coprire il tratto non rilevato (concettualmente il tratto viene ricongiunto, come mostrato in figura 4.5 (b)). In caso contrario è probabile che la fine del tocco fosse intenzionale, e viene quindi generato l'evento *TouchUp*. *IFTouchscreen* mette a disposizione il campo **DragTimeout** per impostare la durata del timer (espressa in millisecondi).

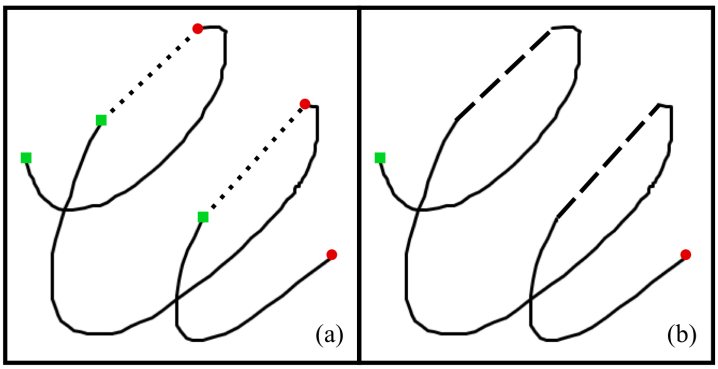


Figura 4.5: Imperfezioni nella rilevazione del tocco e relativa correzione operata da IFTouchscreen

Come nel caso del mouse questa classe consente di regolare la soglia di spostamento oltre la quale attivare il *drag*, grazie al campo **DragDeltaThreshold**.

Infine è possibile scegliere se l'evento Drag deve essere generato solo in corrispondenza di un effettivo spostamento (come avviene per *IFMouse*) oppure ad ogni frame[5]. Questa seconda modalità risulta utile nella realizzazione di alcuni tipi di interazione, ed in particolare quelle implementate dai controller presenti nel framework. È possibile attivare l'una o l'altra modalità settando opportunamente il campo booleano **SendDragWhenIdle**.

4.5 La classe IFKinect

La classe *IFKinect*, che consente la gestione dell'input prodotto dal Microsoft Kinect nei programmi XVR, si basa sull'applicazione **CaveNI** (*CAVE Natural Interaction*) sviluppata nell'ambito di una tesi svolta presso il laboratorio PERCRO [2]. *CaveNI* si occupa di elaborare i dati ricevuti dal Kinect per effettuare il tracciamento delle mani dell'utente ed il riconoscimento di alcune "*gesture*". *CaveNI* definisce un apposito insieme di messaggi per comunicare i risultati dell'elaborazione ad una qualsiasi applicazione (locale o remora) interessata alla loro gestione. I messaggi sono codificati come una sequenza di caratteri e vengono trasmessi in rete su protocollo UDP. Di seguito vengono descritti i principali tipi di messaggio[6] ed il loro formato:

[5] In assenza di spostamento i campi *DeltaX* e *DeltaY* vengono settati a 0.
[6] Sono riportati i soli messaggi gestiti da *IFKinect* e con riferimento alla versione corrente di *CaveNI*.

Messaggio	Descrizione
NEWUSERS, PositionX, PositionY, PositionZ	Viene inviato periodicamente, fintanto che il dispositivo rileva un utente. I parametri rappresentano la posizione dell'utente.
NEWHAND, UserID, HandID	Viene generato quando inizia il tracciamento di una mano. I parametri sono l'identificatore associato all'utente e quello associato alla mano.
LOSTHAND, UserID, HandID	Viene inviato quando la mano non è più tracciata dal dispositivo. I parametri sono gli stessi di *NEWHAND*.
HAND, UserID, HandID, PositionX, PositionY, PositionZ, Grasp	Viene inviato periodicamente e consente di tenere traccia della posizione delle mani. I parametri rappresentano, oltre all'identificatore associato all'utente e quello associato alla mano, la posizione della mano ed il suo stato (aperta o chiusa).
SWIPEUP, **SWIPEDOWN**, **SWIPELEFT**, **SWIPERIGHT**	Messaggi inviati ogni volta che viene riconosciuta una *gesture* di tipo "*swipe*", che consiste in un movimento continuo della mano lungo una delle quattro direzioni cardinali.
WAVE	Viene generato al riconoscimento di una *gesture* di tipo "*wave*", che consiste in una successione di rapidi movimenti della mano da destra a sinistra e da sinistra a destra (come in un saluto).

Attualmente *CaveNI* supporta il rilevamento di un solo utente, quindi il parametro "*UserID*" può essere ignorato. La posizione dell'utente e quella delle sue mani viene espressa in millimetri e nel sistema di riferimento locale del Kinect.

La classe *IFKinect* si occupa della ricezione di questi messaggi e della generazione di un insieme di eventi ad essi correlati, in modo da uniformare l'input prodotto dal Kinect a quello degli altri dispositivi.

Indipendentemente dall'identificatore che *CaveNI* assegna alle mani, *IFKinect* classifica gli eventi in due insiemi: quelli relativi alla mano principale ("*Hand1*") e quelli associati alla mano secondaria ("*Hand2*"). Per mano principale si intende la prima delle due mani ad essere tracciata dal dispositivo. L'utente dovrebbe fare in modo che venga rilevata per prima la mano destra, in modo che questa sia riconosciuta come mano principale.

Gli eventi definiti in questa classe sono:

Hand1Detected / Hand2Detected - Generati quando il dispositivo rileva la mano principale/secondaria.

Hand1Lost / Hand2Lost - Generati quando il dispositivo non riesce più a tracciare una delle due mani.

Hand1Moved / Hand2Moved - Vengono inviati quando l'utente muove una delle due mani.

Hand1StartGrab / Hand2StartGrab - Vengono generati quando l'utente inizia un'operazione di "*grab*" (muove la mano dopo averla chiusa a pugno).

Hand1Grab / Hand2Grab - Generati quando l'utente muove una delle due mani tenendola chiusa.

Hand1EndGrab / Hand2Grab - Vengono generati quando il "*grab*" termina, ovvero quando l'utente apre la mano con la quale stava effettuando un "*grab*".

SwipeUp / SwipeDown / SwipeLeft / SwipeRight - Inviati ad ogni riconoscimento di una gesture di tipo "*swipe*".

Wave - Viene generato al verificarsi di una gesture di tipo "*wave*".

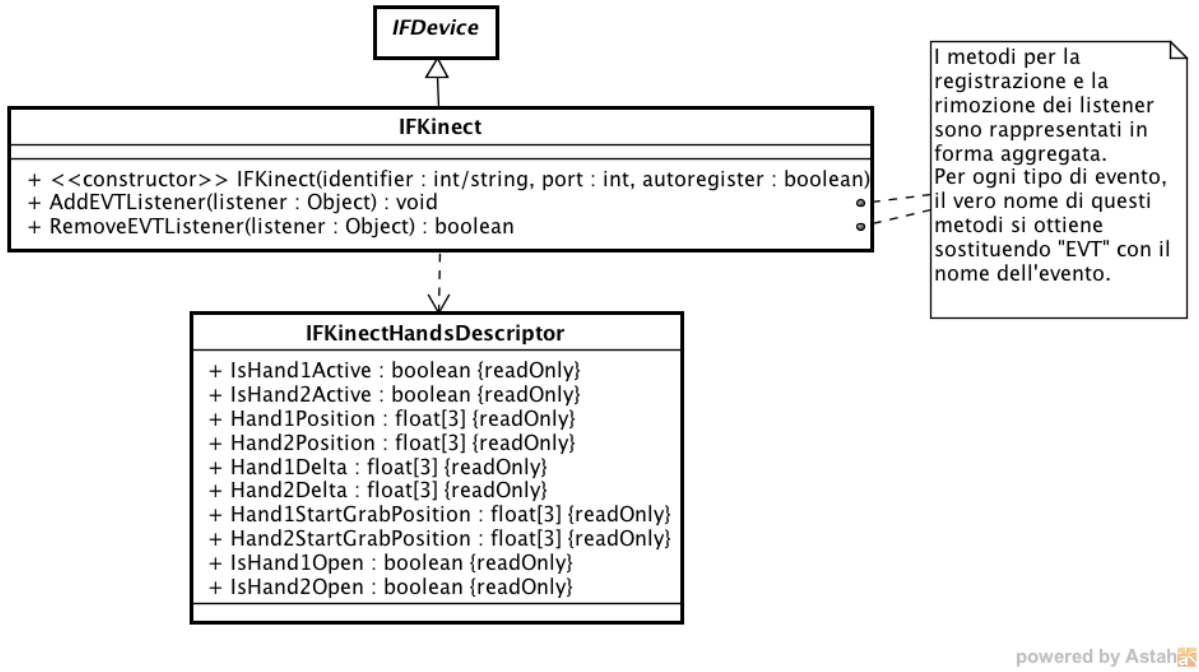


Figura 4.6: Le classi *IFKinect* e *IFKinectHandsDescriptor*

Come sempre, i listener dell'evento EVT devono implementare il metodo

void OnEVT(descriptor)

il cui descrittore è un'istanza della classe *IFKinectHandsDescriptor* (vedi figura 4.6), contenente:

- Lo stato di attivazione delle due mani (ovvero se queste vengono attualmente tracciate o meno).
- La posizione delle stesse.
- L'entità dell'ultimo spostamento.
- La posizione in cui è iniziato il "*grab*" (se è in corso).
- Un campo booleano che indica se la mano considerata è attualmente chiusa o aperta.

È importante notare che, mentre *CaveNI* esprime la posizione delle mani secondo il sistema di riferimento locale al Kinect, *IFKinect* utilizza un sistema di riferimento cartesiano relativo all'utente. Inoltre le coordinate vengono espresse in centimetri (anziché millimetri).

Infine, il costruttore di *IFKinect* necessita un parametro aggiuntivo rispetto alle altre classi che modellano dispositivi. Questo parametro rappresenta la porta sulla quale aprire il socket UDP per la ricezione dei messaggi. Lo stesso numero di porta deve essere impostato nel file di configurazione di *CaveNI*.

4.5.1 IFKinectDrawer

Il framework non si occupa di fornire feedback visivo all'utente, anche se fornisce al programmatore gli strumenti necessari alla sua realizzazione. L'unica eccezione è costituita dalla classe *IFKinectDrawer*, che disegna a schermo un pannello nel quale viene rappresentato (in forma schematica) l'utente e lo stato delle sue mani, in accordo alle rilevazioni del Kinect. L'eccezione è dovuta al fatto che, a causa della scarsa sensibilità e precisione del dispositivo, il Kinect può risultare quasi impossibile da usare senza degli opportuni indicatori visivi.

Il pannello disegnato da *IFKinectDrawer* è mostrato in figura 4.7:

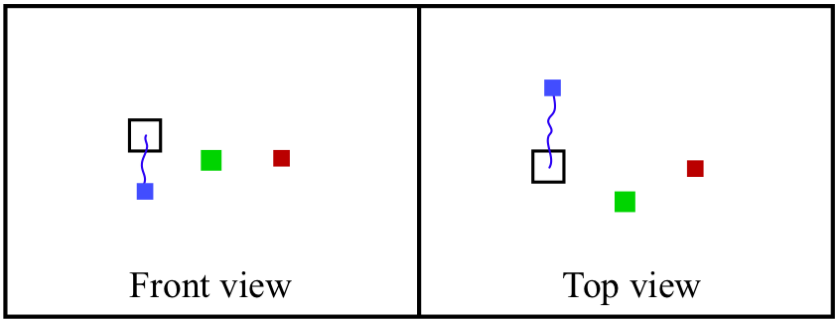


Figura 4.7: Il pannello disegnato da *IFKinectDrawer*

Il quadrato verde rappresenta l'utente, mentre i quadrati rossi e blu indicano rispettivamente la posizione della mano principale e di quella secondaria. Mentre l'utente effettua un "*grab*" viene disegnata una sorta di scia, al fine di evidenziare i movimenti compiuti dalla mano. In corrispondenza del punto in cui l'utente ha chiuso la mano viene disegnato anche un quadrato nero (di dimensioni regolabili), detto "*indicatore del grab*". L'utilità dell'indicatore deriva dal fatto che molti controller attivano le funzionalità associata al *grab* solo quando la posizione della mano si trova ad una certa distanza dal punto di avvio. Segue che l'utente può attivare e disattivare queste funzionalità semplicemente spostando la mano dentro o fuori l'indicatore del grab. La sua dimensione è regolabile per mezzo del campo **GrabIndicatorSize** di *IFKinectDrawer* e dovrebbe essere impostata al valore indicato nella documentazione del controller in uso.

Il pannello viene disegnato in trasparenza, in modo da non nascondere completamente la scena sottostante, e può essere nascosto impostando a *false* il campo **Visible** di *IFKinectDrawer*.

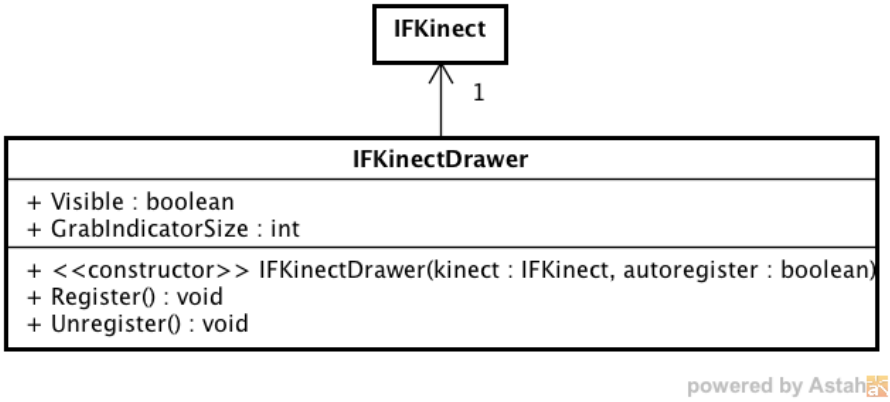


Figura 4.8: La classe *IFKinectDrawer*

4.6 La classe IFContainerPage

XVR consente lo scambio di messaggi tra il player ed il suo contenitore (ad esempio una pagina Web) per mezzo delle funzioni di libreria *DataIn* e *DataOut*. In particolare:

- *DataIn* verifica l'arrivo di un messaggio e, se presente, lo restituisce sotto forma di stringa. In assenza di nuovi messaggi la funzione restituisce una stringa vuota.
- *DataOut* prende come argomento una stringa e la inoltra al contenitore del player.

La classe *IFContainerPage* modella il contenitore del player XVR come un dispositivo che genera eventi di tipo **DataIn**. Il principale scopo di questa classe è quello di uniformare la gestione di questo tipo di input con quanto avviene per gli altri dispositivi. Inoltre si occupa di filtrare i messaggi vuoti, notificando solo quelli significativi.

I listener dell'evento **DataIn** devono implementare il metodo

void OnDataIn(String message)

Il descrittore associato a questo evento è costituito dal messaggio stesso.

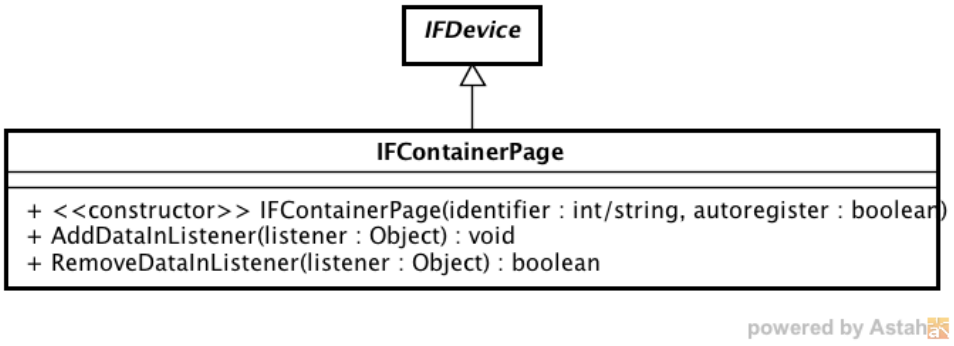


Figura 4.9: La classe *IFContainerPage*

Capitolo 5

Il package "Navigation metaphors"

Nel capitolo 1 è stato introdotto il concetto di *metafora di interazione* e sono stati presentati diversi esempi di metafore per la navigazione all'interno della scena. In questo capitolo ci occuperemo delle classi che modellano queste metafore e delle *tecniche di interazione* implementate dai relativi "controller"[1]. Metafore e controller hanno un buon grado di configurabilità, che ne consente l'uso in un ampio insieme di scenari. Ci sono tuttavia dei casi in cui è necessario ricorrere a soluzioni ad-hoc. In queste situazioni le metafore ed i controller inclusi nel framework possono comunque risultare utili sia in fase di prototipazione, sia come base di partenza per soluzioni personalizzate.

Come vedremo, l'interfaccia esposta dalle varie metafore e dai relativi controller è abbastanza uniforme. Per tale ragione esamineremo nel dettaglio soltanto la prima metafora, limitando la trattazione delle altre ai soli aspetti che le caratterizzano.

5.1 Le classi IFNavigationMetaphor e IFAnimatedNavigationMetaphor

La classe astratta *IFNavigationMetaphor*, mostrata in figura 5.1, dichiara l'interfaccia minimale di ogni metafora di navigazione.

La classe prevede tre metodi riguardanti posizione e direzione della camera:

GetPosition - Restituisce la posizione corrente della camera.

GetDirection - Restituisce la direzione della camera.

SetPosition - Posiziona la camera nel punto desiderato. Parleremo del secondo parametro ("*sendEvent*") più avanti.

[1] Le classi "controller" sono definite nel sotto-package "Controllers".

È naturale chiedersi per quale ragione non è stato dichiarato anche un metodo **SetDirection**. Questa scelta è dovuta al fatto che non tutte le metafore permettono di impostare una direzione generica, a causa dai vincoli imposti dalla metafora stessa. In questi casi è però presente il metodo **SetDirectionLocal** che consente di impostare la direzione in accordo al sistema di riferimento locale usato dalla metafora e/o rispettando i vincoli previsti dalla metafora stessa. Si noti che questi due metodi possono anche coesistere. Per ragioni analoghe alcune metafore definiscono il metodo **SetPositionLocal**.

Uno dei metodi più importanti dichiarati da *IFNavigationMetaphor* è *"Update"*. Il suo scopo è quello di aggiornare lo stato interno della metafora e ricalcolare posizione e direzione della camera di conseguenza. Fintanto che una metafora è in uso, è necessario invocare questo metodo su di essa ad ogni frame. A tal scopo è stato definito un apposito controller, chiamato *IFMetaphorUpdateController*, che esamineremo nel paragrafo 5.2.

L'unico campo previsto è "*Camera*", contenente il riferimento all'oggetto CVmCamera su cui agisce la metafora. Questo campo può essere manipolato solo dalle sottoclassi di *IFNavigationMetaphor.*

Quanto agli eventi, ogni metafora di navigazione ne genera almeno due, *MovementStarted* e *MovementEnded*, che corrispondono rispettivamente all'inizio e alla fine di uno spostamento. Questi eventi vengono generati solo quando, in risposta all'input dell'utente[2], viene modificata la posizione della camera nello spazio. Sono quindi esclusi cambi di direzione e spostamenti causati da animazioni. Ogni listener deve gestire entrambi gli eventi, definendo i metodi

void OnMovementStarted(int/string id)

e

void OnMovementEnded(int/string id)

dove il parametro "id" è l'identificatore della metafora che ha scatenato l'evento. L'aggiunta e la rimozione dei listener avviene per mezzo dei metodi **AddMovementListener** e **RemoveMovementListener**. Si noti che tali metodi sono già definiti in questa classe, quindi le sue sottoclassi non devono preoccuparsi della gestione dei listener. Quanto all'invio degli eventi, nell'implementazione della sottoclasse basta invocare i metodi **SendMovementStarted** e **SendMovementEnded**, che provvederanno a notificare tutti i listener registrati.

È ora possibile comprendere il parametro booleano "*sendEvent*" del metodo *SetPosition*: se presente[3] e settato a *true*, *SetPosition* genererà la coppia di eventi suddetti. Durante la fase di inizializzazione dell'applicazione questo parametro deve essere sempre settato a *false*.

[2]Più precisamente in risposta all'azione dei controller sulla metafora, attraverso i metodi ad essi dedicati. Questo avviene quasi sempre in risposta all'input dell'utente.

[3]È un parametro opzionale.

Infine vediamo alcune regole di base per la definizione di una metafora di navigazione. Ogni sottoclasse concreta di *IFNavigationMetaphor* deve:

- Definire un costruttore che prenda come argomenti (almeno) un oggetto *CVmCamera* e un identificatore, e inizializzare con essi i campi *Camera* e *Id* (ereditato da *IFEventsSource*).
- Implementare i metodi astratti *SetPosition*, *GetPosition*, *GetDirection* e *Update*.
- Implementare i metodi astratti *Register* e *Unregister* della classe *IFEventsSource*.
- Definire altri metodi e campi per modificare posizione e orientamento della camera, in accordo alla semantica e ai vincoli della metafora.
- Ricordarsi di generare gli eventi *MovementStarted/MovementEnded* ogni volta che inizia/termina uno spostamento. A tal fine usare i metodi *SendMovementStarted* e *SendMovementEnded*.

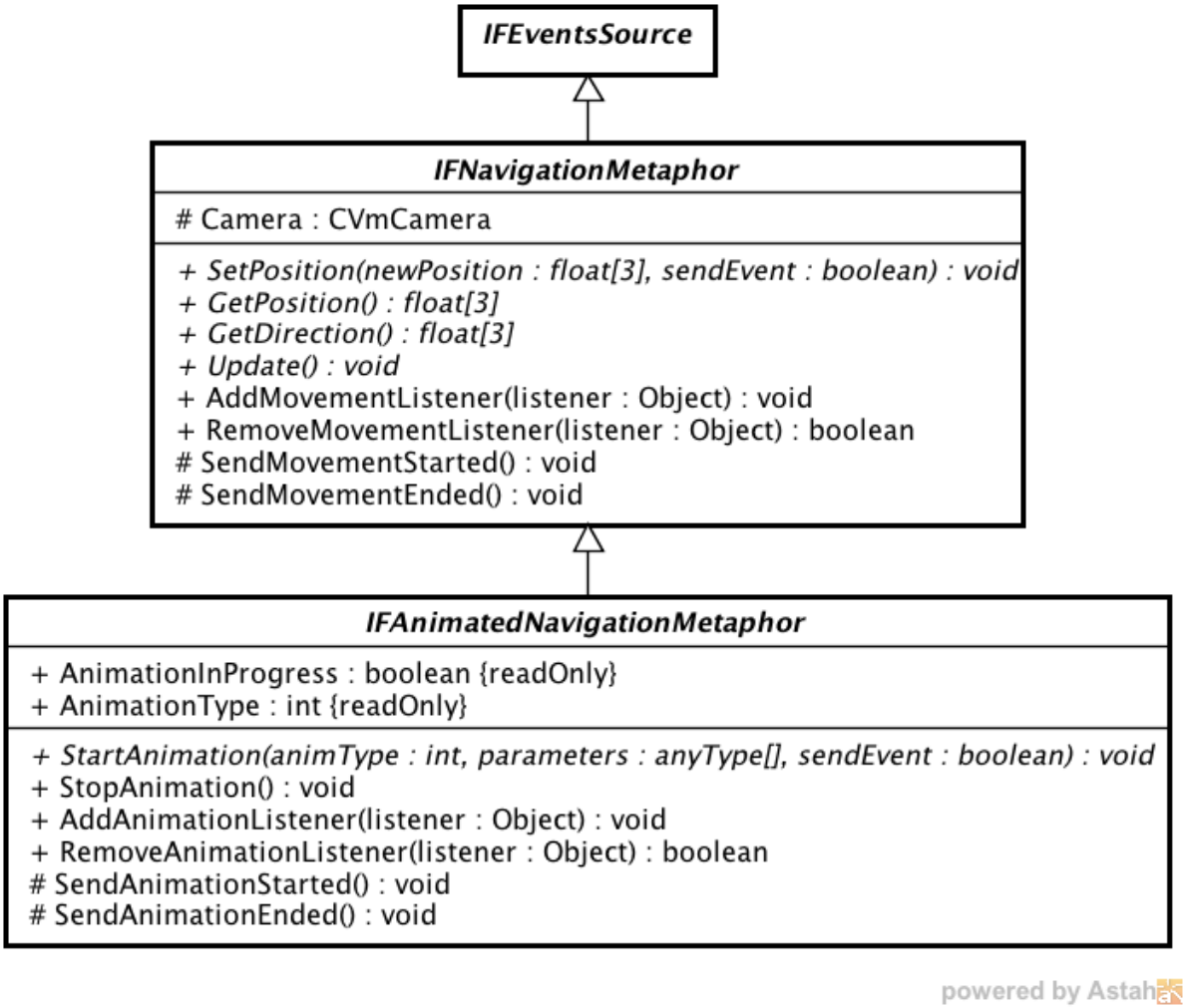


Figura 5.1: Le classi *IFNavigationMetaphor* e *IFAnimatedNavigationMetaphor*

Passiamo ora alla classe astratta *IFAnimatedNavigationMetaphor*. Tutte le metafore di navigazione definite nel framework supportano delle animazioni per modificare, in maniera automatica e graduale, posizione e orientamento della camera. Queste animazioni risultano utili in molti contesti, ad esempio per realizzare una navigazione basata su punti di interesse, oppure per evitare transizioni "brusche" nel passaggio da una metafora ad un'altra che impone vincoli più restrittivi. La classe *IFAnimatedNavigationMetaphor* definisce quindi un'interfaccia comune per il supporto alle animazioni.

Il principale metodo dichiarato in questa classe è **StartAnimation**, che accetta i seguenti argomenti:

animType - Un intero che identifica il tipo di animazione.

parameters - Un array contenente i parametri dell'animazione.

sendEvent - Se impostato a true verranno generati gli eventi *AnimationStarted* e *AnimationEnded* rispettivamente all'inizio e al termine dell'animazione.

Si noti che questo metodo si occupa soltanto di preparare opportunamente lo stato interno della metafora per l'avvio dell'animazione. L'esecuzione vera e propria viene però svolta dal metodo *Update*, che ad ogni invocazione effettua un passo dell'animazione. Una volta in esecuzione, l'animazione può essere interrotta per mezzo del metodo **StopAnimation**.

IFAnimatedNavigationMetaphor definisce anche due campi, *AnimationInProgress* e *AnimationType*, che consentono rispettivamente di conoscere se in un dato istante la metafora sta eseguendo un'animazione ed il tipo della stessa. In particolare, ad *AnimationType* viene assegnato lo stesso valore del parametro *animType* nel metodo *StartAnimation.*

Come accennato, questa classe prevede due ulteriori eventi, *AnimationStarted* e *AnimationEnded*, che vengono generati rispettivamente all'inizio e al termine dell'animazione. Ogni listener deve gestire entrambi gli eventi, definendo i metodi

void OnAnimationStarted(int/string id, int animType)

e

void OnAnimationEnded(int/string id, int animType)

dove il parametro "id" è l'identificatore della metafora che ha scatenato l'evento e *animType* è l'identificatore associato al tipo di animazione. L'aggiunta e la rimozione dei listener avviene per mezzo dei metodi **AddAnimationListener** e **RemoveAnimationListener**.

Infine, nell'implementazione di una sottoclasse concreta di *IFAnimatedNavigationMetaphor* si devono seguire queste semplici regole:

- Dichiarare gli identificatori associati alle animazioni e i parametri richiesti da ognuna di esse.
- Definire il metodo astratto *StartAnimation*, che aggiorna lo stato interno della metafora in modo da prepararla all'avvio dell'animazione. Come minimo l'implementazione deve aggiornare i campi *AnimationInProgress* e *AnimationType*.
- Definire *Update* in modo che, quando *AnimationInProgress* è settato a *true*, esegua un passo dell'animazione richiesta.
- Generare gli eventi *MovementStarted/MovementEnded* ogni volta che inizia/termina un'animazione. A tal fine si devono usare i metodi *SendAnimationStarted* e *SendAnimationEnded*.

5.2 La classe IFMetaphorUpdateController

Uno dei controller più importanti per il corretto funzionamento delle metafore di navigazione è *IFMetaphorUpdateController*. Il suo compito è quello di invocare periodicamente (ad ogni frame) il metodo *Update* delle metafore che gestisce.

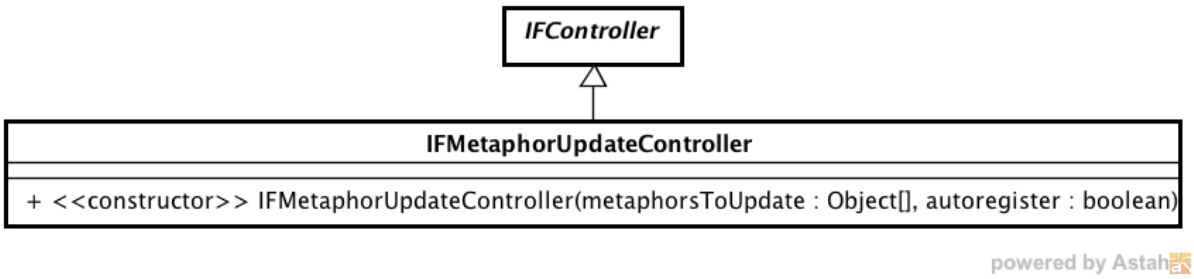


Figura 5.2: La classe *IFMetaphorUpdateController*

Come tutti i controller, esso implementa i metodi definiti dalla classe astratta *IFController*. Poiché abbiamo già esaminato questi metodi nel paragrafo 3.4, e poiché gli oggetti controller espongono tutti la stessa interfaccia (e quindi si utilizzano tutti nello stesso modo), ci limiteremo a descrivere le uniche differenze significative, che consistono nei parametri del costruttore e del metodo *CreateInteractionModality*. Questo approccio alla trattazione dei controller verrà mantenuto anche nel prosieguo della tesi.

I parametri del costruttore sono:

metaphorsToUpdate - Un array di metafore su cui *IFMetaphorUpdateController* invocherà il metodo *Update*. In realtà non esiste un motivo particolare per cui questo controller non possa essere usato anche con altri tipi di metafore o con oggetti di

altra natura. Tuttavia attualmente nel framework le uniche classi che implementano il metodo *Update* sono quelle relative alle metafore di navigazione. Questa è l'unica ragione per cui spesso ci si riferisce a *IFMetaphorUpdateController* solo in questo contesto.

autoregister - Booleano per richiede la registrazione automatica del controller al termine della sua inizializzazione.

Si noti che, in accordo alle specifiche descritte nel paragrafo nel paragrafo 3.4, il costruttore si occupa anche di creare una modalità di interazione di default (il cui identificatore è IFC_CTRL_DEFAULT_INTERMOD_ID), e che le metafore passate come argomento al costruttore vengono in realtà assegnate a tale modalità. Segue che il parametro "*params*" del metodo *CreateInteractionModality* dovrà essere un array dello stesso tipo. Se invece si vuole far sì che la modalità di default sia "disabilitata", basta passare come parametro un array vuoto.

5.3 La classe IFOrbiting

La prima metafora che esamineremo è l'*Orbiting*, implementata dalla classe *IFOrbiting*. Abbiamo già visto questa metafora nel paragrafo 1.5.4. In sostanza, la posizione della camera è vincolata sulla superficie di una sfera centrata in un punto detto "*target*", e la sua direzione è sempre rivolta verso il centro di tale sfera. Da quanto detto segue che, nell'implementazione di questa metafora, è naturale ricorrere ad un sistema di coordinate sferiche.

Tale sistema di coordinate può essere definito a partire da:

- Un'**origine**, nel nostro caso il punto *target T*.
- Due assi ortogonali tra loro e detti rispettivamente **zenith** e **direzione di riferimento azimutale**. Nel caso specifico lo zenith corrisponde all'asse globale Y e la direzione di riferimento azimutale corrisponde all'asse globale Z.

È possibile anche individuare un **piano di riferimento** ortogonale allo zenith e passante per l'origine.

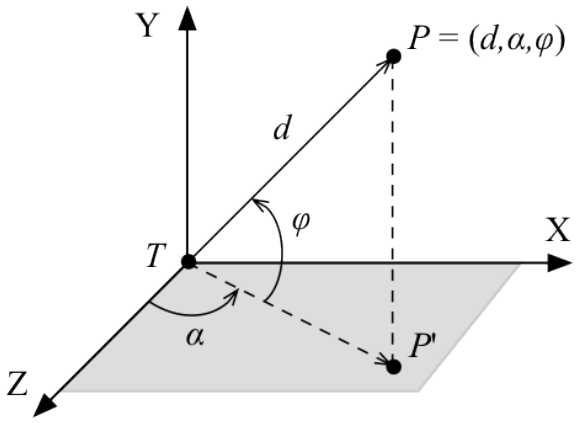


Figura 5.3: Coordinate sferiche

Una volta stabilito il sistema di riferimento, la posizione nello spazio di un punto P può essere definita per mezzo delle seguenti coordinate sferiche:

- La **distanza** d dall'origine, che rappresenta anche il raggio della sfera.
- L'angolo di **elevazione** φ compreso tra il piano di riferimento e il vettore $\overrightarrow{TP}$
- L'angolo di **azimut** α compreso tra la direzione di riferimento azimutale ed il vettore $\overrightarrow{TP'}$, dove P' è la proiezione di P sul piano di riferimento.

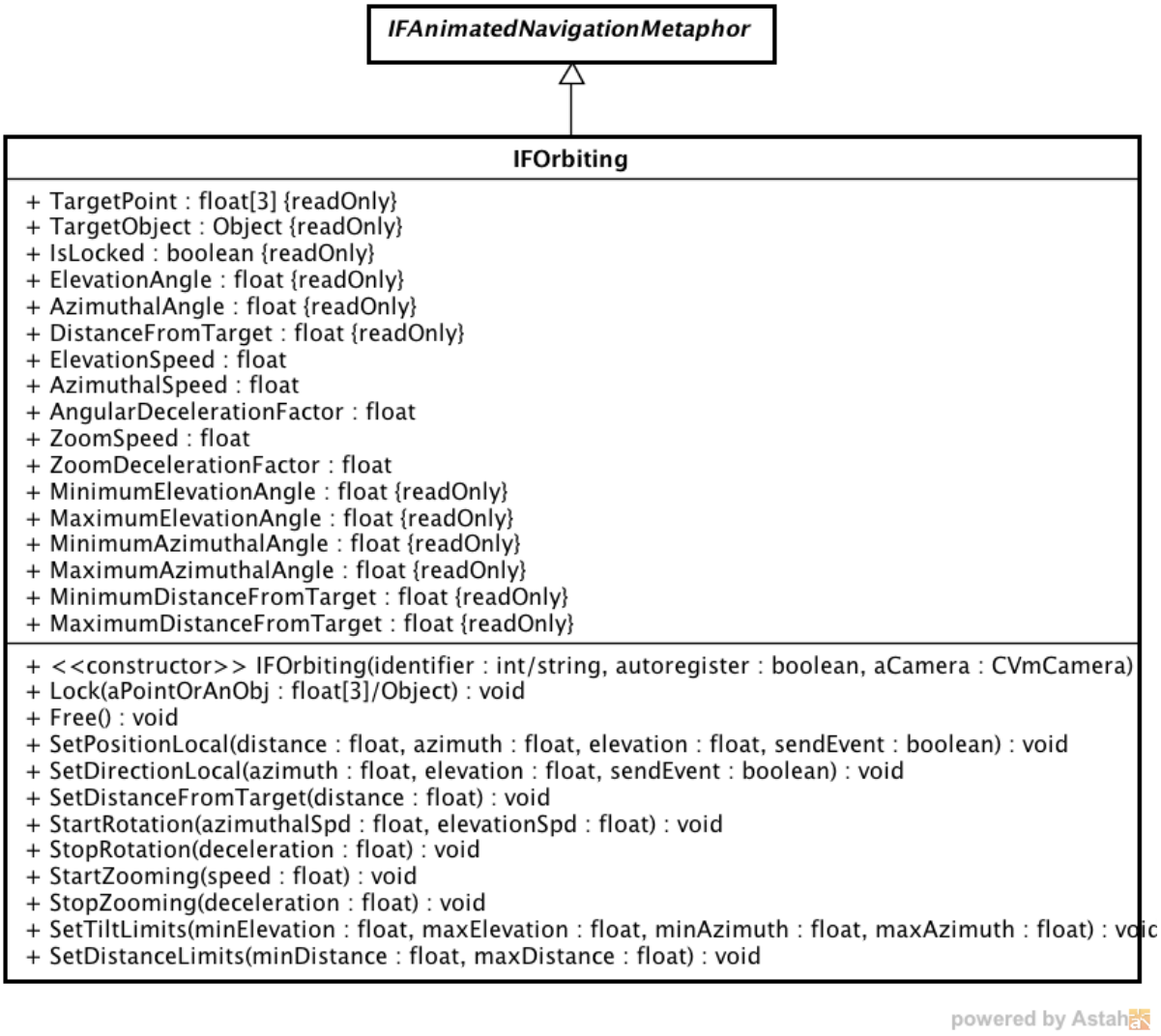


Figura 5.4: La classe *IFOrbiting*

Passiamo ora ad esaminare campi e metodi della classe *IFOrbiting*, classificandoli in base alla loro funzione.

Costruttore

Il costruttore accetta tre parametri:

identifier - È l'identificatore associato alla metafora. Può essere un intero o una stringa.

autoregister - Flag booleano usato per attivare la registrazione automatica dell'oggetto al termine della sua inizializzazione. La registrazione consiste semplicemente nell'inserimento della metafora nella tabella delle fonti di eventi gestita da *IFApplication*.

aCamera - L'oggetto camera gestito dalla metafora. È un parametro opzionale. Se non viene specificato verrà usata la camera attiva.

Si noti che questi parametri sono standard per tutti i costruttori delle metafore di navigazione. Per tale ragione non verranno più discussi nel seguito.

Target point

Una volta creato un nuovo oggetto *IFOrbiting*, è necessario impostare il *target point* per mezzo del metodo **Lock**. L'unico parametro può essere sia un punto[4] nello spazio sia un qualsiasi oggetto che implementa il metodo *GetPosition()*. Nel caso in cui si scelga di passare un oggetto come parametro, non solo la camera sarà sempre orientata verso di esso, ma ne seguirà anche gli spostamenti. Questo consente, tra le altre cose, di utilizzare questa metafora anche per realizzare semplici visuali in terza persona.

Free è il metodo opposto, che rimuove i vincoli sull'orientamento della camera. Si noti che, quando la metafora non si trova in stato di "lock", tutte le chiamate al metodo *Update* "ritornano" immediatamente. Di contro, è necessario "sbloccare" la metafora per mezzo del metodo *Free* ogni volta che si intende attivare un'altra metafora o gestire la camera manualmente.

I campi correlati con queste funzionalità sono:

TargetPoint - Contiene le coordinate del punto *target*, espresse nel sistema di riferimento globale. Se si è scelto di legare la camera ad un oggetto, questo campo verrà costantemente aggiornato con la posizione dell'oggetto stesso. È accessibile in **sola lettura**.

TargetObject - Contiene un riferimento all'oggetto target, se presente. È accessibile in **sola lettura**.

IsLocked - Booleano che indica se la metafora è in stato di "lock" o meno. È accessibile in **sola lettura**.

[4]Rappresentato mediante un array di tre elementi di tipo float. Si noti che in S3D questo genere di array (dimensione fissa e composti da soli *float*) corrispondono al tipo di dato *Vector*. Al contrario, il tipo di dato *Array* del linguaggio S3D consente di creare array dinamici contenenti elementi di qualsiasi tipo (in maniera analoga ai *Vector* di Java e C++).

Posizione e direzione della camera

Le coordinate sferiche che esprimono la posizione della camera vengono memorizzate nei campi **ElevationAngle**, **AzimuthalAngle** e **DistanceFromTarget**, tutti accessibili in sola lettura. Per ottenere la posizione e direzione espresse nel sistema di riferimento globale, basta ricorrere rispettivamente ai metodi **GetPosition** e **GetDirection**.

La posizione della camera può essere invece impostata grazie ai metodi **SetPosition** e **SetPositionLocal**. La differenza tra i due consiste nel fatto che il primo richiede che la posizione venga espressa nel sistema di riferimento globale, mentre nel secondo le coordinate vengono espresse nel sistema di riferimento sferico locale. È infine possibile modificare i soli angoli di elevazione e azimut tramite **SetDirectionLocal**, oppure la sola distanza dal target grazie a **SetDistanceFromTarget**. Tutti questi metodi accettano "*sendEvent*" come ulteriore parametro opzionale, per abilitare la generazione degli eventi *MovementStarted/MovementEnded*.

Velocità angolari e zoom

Tipicamente la posizione della camera non viene impostata direttamente, bensì settando velocità e accelerazione della stessa. Questa classe fornisce i seguenti campi per tale scopo (modificabili direttamente):

AzimuthalSpeed - Componente della velocità orbitale riferita al solo azimut, espressa in radianti per frame.

ElevationSpeed - Componente della velocità orbitale riferita al solo angolo di elevazione, espressa in radianti per frame.

ZoomSpeed - Velocità di variazione della distanza dal *target*, espressa in unità per frame.

Ad ogni invocazione del metodo *Update*, il valore contenuto in questi campi viene utilizzato per modificare la posizione della camera. Per fermare uno spostamento, basta impostare tutti e tre i campi a 0. Tuttavia è possibile ottenere un arresto più graduale mediante degli opportuni fattori di decelerazione:

AngularDecelerationFactor - Fattore di decelerazione per *AzimuthalSpeed* ed *ElevationSpeed*. Deve assumere un valore reale compreso nell'intervallo [0,1].

ZoomDecelerationFactor - Fattore di decelerazione per *ZoomSpeed*. Deve assumere un valore reale compreso nell'intervallo [0,1].

Ad ogni invocazione del metodo *Update*, i campi di velocità vengono moltiplicati per il valore contenuto in questi campi. Segue che durante uno spostamento il loro valore deve essere settato a 1. Per produrre un arresto graduale, basta invece assegnare a questi campi un valore compreso tra 0 e 1 (esclusi). Ogni classe dichiara a tal fine delle costanti contenenti il valore di default dei fattori di decelerazione. Nel caso di IFOrbiting, queste costanti sono IFC_ORB_DEFAULT_ANGULAR_DECELERATION_FACTOR e IFC_ORB_DEFAULT_ZOOM_DECELERATION_FACTOR. Infine, se si vuole ottenere un arresto improvviso, si devono settare questi campi a 0, senza la necessità di agire sulle velocità.

Vengono inoltre forniti i seguenti metodi per semplificare la gestione dei suddetti campi:

void StartRotation(**float azimuthalSpd,** **float elevationSpd)**	Avvia l'orbiting o, se già in corso, ne modifica le velocità correnti. Di fatto è equivalente al codice seguente: *AzimuthalSpeed = azimuthalSpd;* *ElevationSpeed = elevationSpd;* *AngularDecelerationFactor = 1.0;*
void StopRotation(**float deceleration)**	Arresta l'orbiting. È equivalente al codice seguente: *AngularDecelerationFactor = deceleration;*
void StartZooming(**float speed)**	Avvia lo "zoom" o, se già in corso, ne modifica la velocità corrente. È equivalente al codice seguente: *ZoomSpeed = speed;* *ZoomDecelerationFactor = 1.0;*
void StopZooming(**float deceleration)**	Arresta lo "zoom". È equivalente al codice seguente: *ZoomDecelerationFactor = deceleration;*

Limiti sul posizionamento

È possibile imporre dei limiti sulla posizione della camera. Tali limiti vengono imposti separatamente su ogni coordinata e sono memorizzati nei seguenti campi (accessibili in sola lettura):

MinimumElevationAngle - Valore minimo che può assumere *ElevationAngle*.

MaximumElevationAngle - Valore massimo che può assumere *ElevationAngle*.

MinimumAzimuthalAngle - Valore minimo che può assumere *AzimuthalAngle*.

MaximumAzimuthalAngle - Valore massimo che può assumere *AzimuthalAngle*.

MinimumDistanceFromTarget - Valore minimo che può assumere *DistanceFromTarget*.

MaximumDistanceFromTarget - Valore massimo che può assumere *DistanceFromTarget*.

Tali limiti non possono essere imposti modificando direttamente i suddetti campi, bensì facendo uso dei seguenti metodi:

void SetElevationLimits(float minElevation, float maxElevation)

void SetAzimuthLimits(float minAzimuth, float maxAzimuth)

void SetDistanceLimits(float minDistance, float maxDistance)

Questo è necessario per assicurare che i valori richiesti non superino dei limiti globali definiti per garantire il corretto funzionamento della metafora e/o per evitare orientamenti che possono infastidire e disorientare l'utente. In particolare:

- I valori assegnabili a *MinimumElevationAngle* e *MaximumElevationAngle* devono essere sempre compresi nell'intervallo [-85º, 85º], definito per mezzo delle costanti IFC_ORB_MAX_ELEVATION e IFC_ORB_MIN_ELEVATION.
- *MinimumDistanceFromTarget* non può assumere valori inferiori alla distanza tra la camera ed il *near plane*.
- *MaximumDistanceFromTarget* non può assumere valori superiori alla distanza tra la camera ed il *far plane*.

Animazioni

La classe *IFOrbiting* supporta le seguenti animazioni per il riposizionamento della camera:

IFC_ORB_ANIMATED_ORBITING_BY_ANGLES - Riposiziona la camera fornendo le nuove coordinate espresse nel sistema di riferimento sferico locale. L'argomento *"parameters"* del metodo *StartAnimation* è un array contenente la nuova elevazione, il nuovo azimut e la nuova distanza dal target.

IFC_ORB_ANIMATED_ORBITING_BY_POSITION - Riposiziona la camera fornendo le nuove coordinate espresse nel sistema di riferimento globale. L'argomento *"parameters"* consiste stavolta in un array contenente, come unico argomento, le nuove coordinate della camera.

5.3.1 I controller di IFOrbiting

Vediamo ora le classi relative alle *tecniche di interazione* per la metafora "*Orbiting*". Per ognuna di esse verrà fornita una breve descrizione su come la tecnica implementata consente di controllare la metafora e quali sono i parametri che ne definiscono il comportamento.

Keyboard controller

Una tecnica di interazione per la tastiera viene fornita dalla classe *IFOrbitingKeyboardController*. Il tipo di interazione implementata è abbastanza semplice: si usano delle una coppie di tasti per incrementare e ridurre ognuna delle tre coordinate sferiche.

I parametri del costruttore sono:

1. **keyboard** - Una istanza di *IFKeyboard* con la quale si vuole controllare la metafora.
2. **autoregister** - Flag booleano per richiede la registrazione automatica del controller al termine della sua inizializzazione.
3. **orbiting** - La metafora da controllare (istanza di *IFOrbiting*)
4. **orbitUpButton** - Il codice associato al pulsante usato per incrementare l'elevazione.
5. **orbitDownButton** - Il codice associato al pulsante usato per decrementare l'elevazione.
6. **orbitRightButton** - Il codice associato al pulsante usato per incrementare l'azimut.
7. **orbitLeftButton** - Il codice associato al pulsante usato per decrementare l'azimut.
8. **zoomInButton** - Il codice associato al pulsante usato per decrementare la distanza dal target.
9. **zoomOutButton** - Il codice associato al pulsante usato per incrementare la distanza dal target.
10. **rotationSpeed** - Parametro opzionale che consente di regolare la velocità di rotazione. Il suo valore di default è definito dalla costante IFC_ORBKC_DEFAULT_ROTATION_SPEED.
11. **zoomSpeed** - Parametro opzionale che consente di regolare la velocità di avvicinamento/allontanamento dal target. Il suo valore di default è definito dalla costante IFC_ORBKC_DEFAULT_ZOOM_SPEED.

È anche possibile evitare di assegnare uno o più pulsanti passando ai relativi parametri la costante IFC_CTRL_UNASSIGNED oppure, più semplicemente, assegnandoli a NULL. È inoltre possibile far sì che la modalità di default sia "disabilitata" passando al costruttore soltanto i primi tre argomenti.

Quanto al metodo *CreateInteractionModality*, gli elementi da passare come parametro sono un sottoinsieme degli argomenti del costruttore, ed in particolare quelli numerati da 4 a 11 (fermo restando la possibilità di omettere gli argomenti opzionali).

Mouse controller

La classe *IFOrbitingMouseController* permette di controllare la metafora orbiting tramite mouse. L'utente può orbitare attorno ad un punto di interesse semplicemente "draggando" con il mouse nella direzione desiderata. Un drag verticale con un altro tasto attiva invece la modalità di zoom. Un aspetto da notare è che la velocità di spostamento in un dato istante non è proporzionale all'entità dell'ultimo spostamento del mouse, bensì alla distanza tra la posizione corrente del mouse ed il punto in cui è iniziato il drag. Questo approccio, che è stato adottato per tutti i "mouse controller" per metafore di navigazione, fa sì che l'utente non debba muovere costantemente il mouse per continuare la rotazione (o lo spostamento). *IFOrbitingMouseController* consente inoltre di sfruttare la "rotella" del mouse per avvicinarsi o allontanarsi dal *target*.

Il costruttore di questo controller accetta i seguenti parametri:

1. **mouse** - Una istanza di *IFMouse* con la quale si vuole controllare la metafora.
2. **autoregister** - Flag booleano per richiede la registrazione automatica del controller al termine della sua inizializzazione.
3. **orbiting** - La metafora da controllare (istanza di *IFOrbiting*)
4. **orbitButton** - Il bottone del mouse da usare per ruotare attorno al *target*.
5. **zoomButton** - Il bottone del mouse da usare per avvicinarsi o allontanarsi dal *target*.
6. **useWheelForZoom** - Flag booleano per abilitare l'uso della rotella del mouse per effettuare lo zoom.
7. **zoomWheelSpeed** - Parametro opzionale che consente di regolare la sensibilità della rotella del mouse. Il suo valore di default è definito dalla costante IFC_ORBKC_DEFAULT_ZOOM_SPEED.

8. **rotationSensibility** - Parametro opzionale che consente di regolare la velocità di rotazione. Il suo valore di default è definito dalla costante IFC_ORBMC_DEFAULT_ROTATION_SENSIBILITY.

9. **zoomSensibility** - Parametro opzionale che consente di regolare la velocità di avvicinamento/allontanamento rispetto al *target*. Il suo valore di default è definito dalla costante IFC_ORBMC_DEFAULT_ZOOM_SENSIBILITY.

I parametri 4 e 5 possono assumere i valori IFC_MOUSE_RIGHT_BUTTON, IFC_MOUSE_LEFT_BUTTON, IFC_MOUSE_MIDDLE_BUTTON oppure, se non si intende assegnare alcun pulsante, IFC_CTRL_UNASSIGNED o NULL.

Per quanto riguarda il metodo *CreateInteractionModality*, devono essere forniti i parametri da 4 a 9.

Touchscreen controller

La classe *IFOrbitingTouchController* implementa una *tecnica di interazione* per controllare la metafora *Orbiting* attraverso il touchscreen. L'idea alla base di questa tecnica è quella di suddividere la finestra in due aree: un'area principale (che occupa gran parte dello schermo) ed un "margine attivo", come mostrato in figura 5.5. L'utente può orbitare attorno al punto di interesse "draggando" nell'area principale[5]. Draggando lungo il margine, invece, si modifica la distanza tra la camera ed il punto di interesse (*zoom*). È importante notare che, a differenza della tecnica di interazione implementata per il mouse, la velocità è proporzionale all'entità dell'ultimo spostamento del cursore, indipendentemente dal punto in cui è iniziato il drag. Questo approccio, utilizzato anche negli altri controller per touchscreen, è infatti quello più diffuso nella pratica, e quindi più vicino alle aspettative degli utenti. Infine, i fattori di decelerazione impiegati in tutti i controller per touchscreen hanno un valore più basso rispetto a quelli di default, al fine di ottenere l'effetto noto come ***kinetic scrolling***.

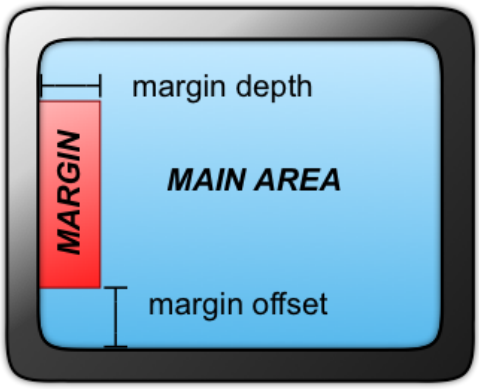


Figura 5.5: Margine "attivo"

Passando ai parametri del costruttore, oltre ai soliti primi tre relativi al dispositivo, al flag di auto-registrazione e alla metafora, i più significativi sono:

[5]In realtà basta che il drag inizi nell'area principale. Una volta in corso, l'utente può "invadere" anche il margine, senza che l'azione corrente venga interrotta.

4. **margin** - Consente di scegliere dove collocare il margine attivo. Accetta uno dei seguenti valori: IFC_ORBTC_NO_ZOOM_ACTIVE_MARGIN, IFC_ORBTC_ZOOM_ACTIVE_MARGIN_TOP, IFC_ORBTC_ZOOM_ACTIVE_MARGIN_BOTTOM, IFC_ORBTC_ZOOM_ACTIVE_MARGIN_LEFT, IFC_ORBTC_ZOOM_ACTIVE_MARGIN_RIGHT.

5. **marginOffset** - La distanza tra l'angolo della finestra ed il margine attivo, espressa in pixel.

6. **marginDepth** - La lunghezza del margine.

7. **rotationSensibility** - Parametro opzionale che consente di regolare la velocità di rotazione. Il suo valore di default è definito dalla costante IFC_ORBTC_DEFAULT_ROTATION_SENSIBILITY.

8. **zoomSensibility** - Parametro opzionale che consente di regolare la velocità di avvicinamento/allontanamento rispetto al *target*. Il suo valore di default è definito dalla costante IFC_ORBTC_DEFAULT_ZOOM_SENSIBILITY

Come sempre i parametri dell'elenco sono gli stessi richiesti dal metodo *CreateInteractionModality*. Inoltre, come avviene per la maggior parte dei controller definiti nel framework, è possibile creare un oggetto *IFOrbitingTouchController* senza passare questi parametri al costruttore. Così facendo si ottiene una modalità di interazione di default "disabilitata".

Si noti infine che, come già spiegato, questo framework non si occupa direttamente di fornire feedback visivo[6]. Segue che il codice necessario a disegnare il margine attivo è a carico del programmatore.

Kinect controller

L'ultimo controller associato a *IFOrbiting* è *IFOrbitingKinectController*. Per descrivere il modo in cui l'utente può interagire con la metafora usando il Kinect, ci serviremo di alcune semplici illustrazioni, come quella mostrata in figura 5.6. In queste illustrazioni vengono raffigurati i movimenti che deve eseguire l'utente per attivare le varie funzionalità della metafora. Attualmente l'interazione tramite Kinect è guidata dalle sole mani, per cui ci concentreremo soltanto sul loro stato (aperte o chiuse) e sul modo in cui muoverle. La mano principale[7] viene raffigurata in figura con un cerchio rosso, la secondaria con un cerchio blu. Se il cerchio ha un riempimento bianco significa che la mano deve essere tenuta aperta, mentre se è colorato deve essere tenuta chiusa. Per ottenere buoni risultati

[6] Le uniche eccezioni sono la classe *IFKinectDrawer* e (ovviamente) la GUI.

[7] Ovvero la mano che per prima viene rilevata dal Kinect. I destrimani dovrebbero usare la mano destra come mano principale, i mancini la sinistra.

si consiglia di mantenere i palmi (se la mano è aperta) o i pugni (se è chiusa) orientati verso il Kinect.

Tornando a *IFOrbitingKinectController*, l'interazione avviene come segue:

- Tenendo chiusa la mano principale e spostandola in verticale e in orizzontale, la camera inizia ad orbitare attorno al punto di interesse.
- Tenendo chiusa la mano secondaria e spostandola in avanti o indietro, la camera sia avvicina/allontana al/dal punto di interesse.

È importante notare che in entrambi i casi la velocità è proporzionale alla distanza sta il punto di inizio del "grab" e la posizione attuale della mano, in analogia a quanto avviene per il mouse e a differenza del touchscreen. Segue che il movimento può essere fermato (oltre che aprendo il palmo) riportando la mano in una zona prossima al punto di partenza. Per agevolare questa operazione si consiglia l'uso della classe *IFKinectDrawer* (vista nel paragrafo 4.5.1), che fornisce un apposito indicatore visivo per evidenziare questa zona.

La scelta di questo approccio è motivata sia dalla necessità di non far stancare troppo l'utente, sia dalla scarsa affidabilità del dispositivo (cosa che pregiudica ogni alternativa). Per tali ragioni questo approccio è stata applicato a tutti i "kinect controller" per metafore di navigazione.

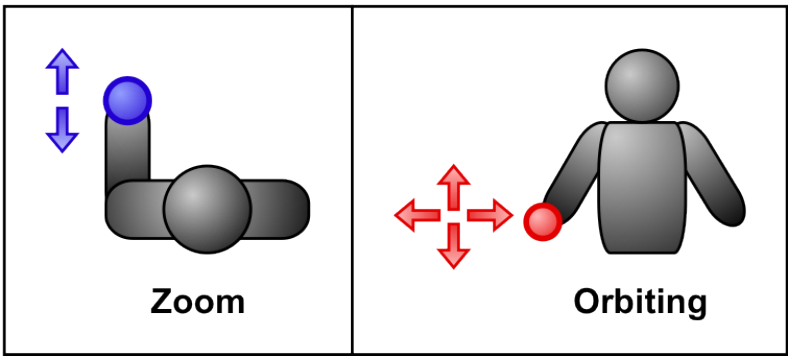


Figura 5.6: Uso del Microsoft Kinect per controllare la metafora *Orbiting*

Quanto ai parametri del costruttore, i più importanti sono:

4. **enabled** - Flag opzionale che, se impostato a *false* o assente, fa sì che la modalità di default sia "disabilitata".
5. **rotationSensibility** - Parametro opzionale che consente di regolare la velocità di rotazione. Il suo valore di default è definito dalla costante IFC_ORBMKC_DEFAULT_ROTATION_SENSIBILITY.

6. **zoomSensibility** - Parametro opzionale che consente di regolare la velocità di avvicinamento/allontanamento rispetto al *target*. Il suo valore di default è definito dalla costante IFC_ORBMKC_DEFAULT_ZOOM_SENSIBILITY.

I parametri del metodo *CreateInteractionModality* comprendono invece solo gli ultimi due argomenti del costruttore (se necessari, sono opzionali).

5.4 La classe IFCylindricalOrbiting

In questa metafora, molto simile all'Orbiting, la camera è vincolata sulla superficie di un cilindro, mentre la sua direzione è sempre rivolta verso l'asse longitudinale dello stesso.

La classe *IFCylindricalOrbiting* utilizza un sistema di riferimento cilindrico per esprimere la posizione della camera. Tale sistema di coordinate può essere definito stabilendo un'**origine**, l'**asse longitudinale del cilindro** e una **direzione di riferimento azimutale**. Da questi è poi possibile ricavare un **piano di riferimento** passante per l'origine e ortogonale all'asse del cilindro. Una volta stabilito il sistema di riferimento, la posizione nello spazio di un punto P può essere definita per mezzo delle seguenti coordinate cilindriche:

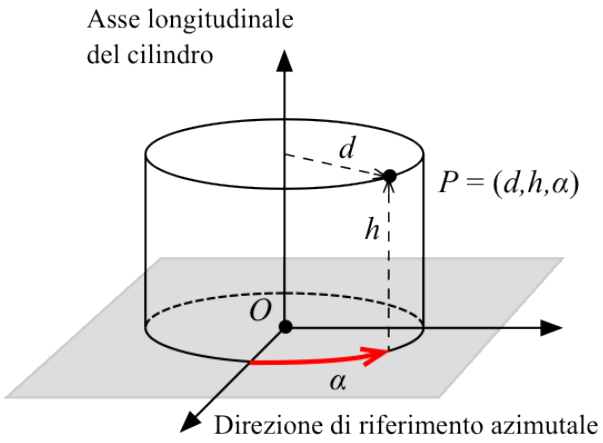


Figura 5.7: Coordinate cilindriche

- La **distanza *d*** dall'asse longitudinale, che rappresenta anche il raggio del cilindro.
- L'**altezza *h*** tra il piano di riferimento e il punto.
- L'angolo di **azimut α** compreso tra la direzione di riferimento azimutale e la proiezione di P sul piano di riferimento.

Vediamo ora i campi ed i metodi implementati dalla classe *IFCylindricalOrbiting*, classificandoli in base alla loro funzione.

Inizializzazione

Dopo aver creato un oggetto *IFCylindricalOrbiting*, è necessario invocare il metodo **Lock**. Come nel caso della classe *IFOrbiting*, tale metodo serve per inizializzare il sistema di riferimento usato dalla metafora e per vincolare la direzione della camera all'asse del cilindro. Tuttavia in questo caso il metodo *Lock* prevede tre parametri:

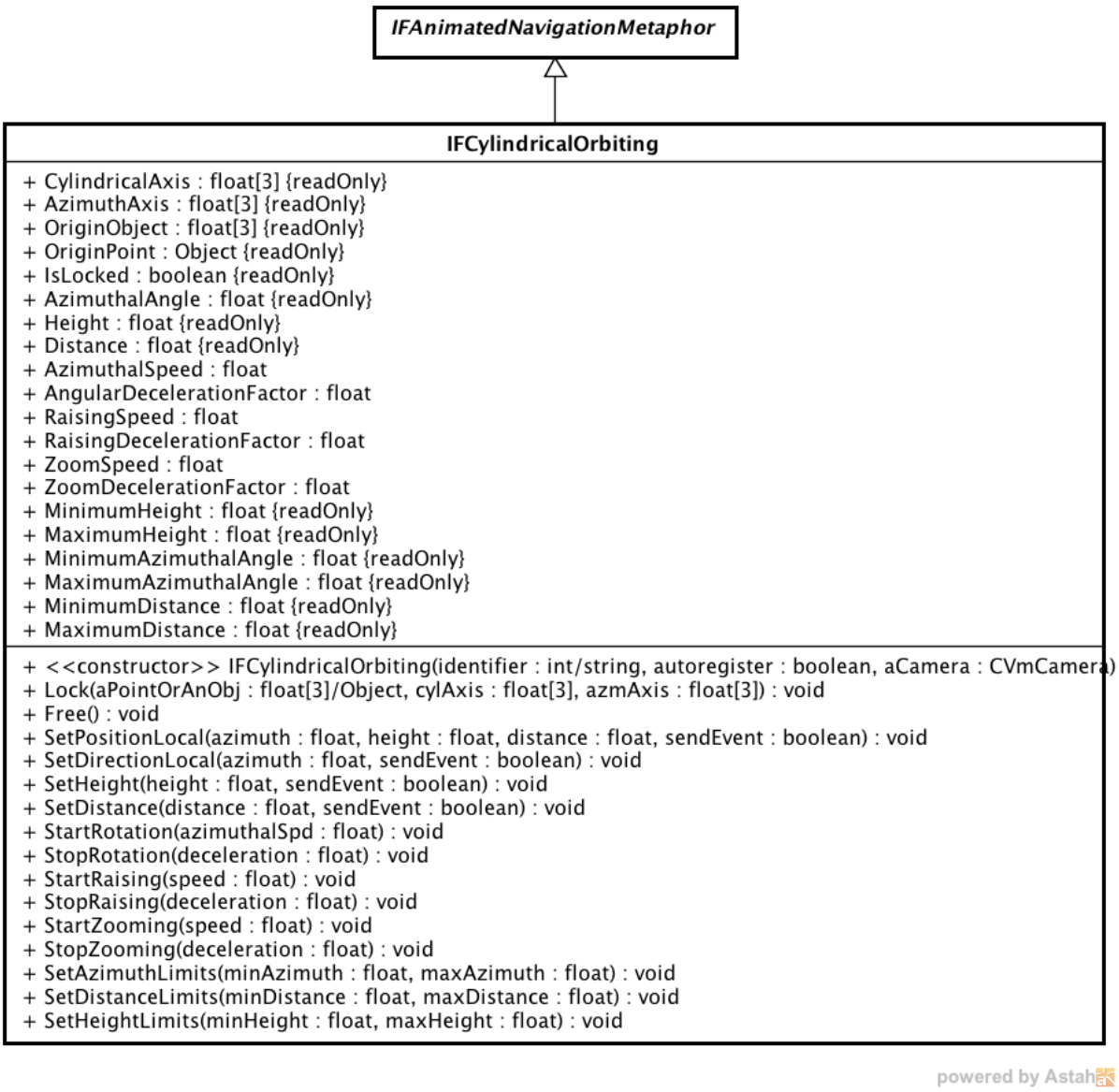


Figura 5.8: La classe *IFCylindricalOrbiting*

1. L'origine del sistema di coordinate espressa come un punto o come un oggetto che implementa il metodo *GetPosition()*. Se viene passato un oggetto la camera ne seguirà gli spostamenti.

2. L'asse longitudinale del cilindro. Infatti questa metafora consente di orbitare attorno ad un asse generico, e non soltanto attorno all'asse Y globale.

3. Un asse azimutale di riferimento. Deve necessariamente essere ortogonale all'asse del cilindro.

Anche stavolta è presente un metodo **Free** per rimuovere i vincoli sull'orientamento della camera.

I campi relativi a queste funzionalità (tutti accessibili in sola lettura) sono:

CylindricalAxis - Memorizza l'asse longitudinale del cilindro, espresso nel sistema di riferimento globale.

AzimuthAxis - Contiene l'asse azimutale di riferimento, espresso nel sistema di riferimento globale.

OriginPoint - Contiene le coordinate dell'origine del sistema di riferimento locale. Se si è scelto un oggetto come origine, questo campo verrà costantemente aggiornato con la posizione dell'oggetto stesso.

OriginObject - Contiene un riferimento all'oggetto da usare come origine del sistema di riferimento locale, se presente.

IsLocked - Booleano che indica se la metafora è in stato di "lock" o meno.

Posizione e direzione della camera

In maniera analoga a *IFOrbiting*, la classe *IFCylindricalOrbiting* mette a disposizione degli appositi campi per ottenere la posizione della camera in coordinate cilindriche. Tali campi sono **AzimuthalAngle**, **Height** e **Distance**. Ovviamente è sempre possibile ottenere posizione e direzione espresse nel sistema di riferimento locale per mezzo dei metodi **GetPosition** e **GetDirection**. La posizione della camera può essere impostata tramite **SetPosition** e **SetPositionLocal**. Per modificare singolarmente le coordinate cilindriche sono disponibili i metodi **SetDirectionLocal** (per l'azimut), **SetDistance** e **SetHeight**.

Anche questa classe consente di limitare la posizione della camera all'interno di precisi intervalli di valori. Gli intervalli possono essere settati per mezzo dei metodi **SetAzimuthLimits**, **SetDistanceLimits** e **SetHeightLimits**, e sono memorizzati nei campi **Minimum**/**MaximumAzimuth**, **Minimum**/**MaximumDistance** e **Minimum**/**MaximumHeight**.

Velocità

Le velocità relative alle tre coordinate cilindriche possono essere impostate modificando i campi **AzimuthalSpeed**, **RaisingSpeed** e **ZoomSpeed**. Ad ognuna di queste corrisponde un fattore di decelerazione che consente di ottenere un arresto graduale e uniforme del moto e, nel caso dei controller per touchscreen, di realizzare l'effetto noto come "*kinetic scrolling*".

Come sempre vengono forniti alcuni metodi di utilità che semplificano la gestione di questi campi: **Start**/**StopRotation**, **Start**/**StopZooming** e **Start**/**StopRaising**.

Animazioni

Le animazioni supportate da *IFCylindricalOrbiting* sono:

IFC_CORB_ANIMATED_ORBITING_BY_COORDS- Riposiziona la camera esprimendo le nuove coordinate nel sistema di riferimento cilindrico. L'argomento *"parameters"* del metodo *StartAnimation* è un array contenente il nuovo azimut, la nuova altezza e la nuova distanza dal target.

IFC_CORB_ANIMATED_ORBITING_BY_POSITION- Riposiziona la camera esprimendo le nuove coordinate nel sistema di riferimento globale. L'unico elemento dell'array *"parameters"* consiste quindi in un vettore[8] contenente le nuove coordinate della camera.

5.4.1 I controller di IFCylindricalOrbiting

Keyboard controller

L'interazione mediante tastiera è possibile grazie alla classe *IFCylindricalOrbitingKeyboardController*. I principali parametri del suo costruttore sono:

4. **moveUpButton** - Il pulsante per aumentare l'altezza.
5. **moveDownButton** - Il pulsante per ridurre l'altezza.
6. **orbitRightButton** - Il pulsante per "orbitare" verso destra.
7. **orbitLeftButton** - Il pulsante per "orbitare" verso sinistra.
8. **zoomInButton** - Il pulsante che attiva lo "zoom-in".
9. **zoomOutButton** - Il pulsante che attiva lo "zoom-out".
10. **rotationSpeed** - Parametro opzionale che consente di regolare la velocità di rotazione. Il suo valore di default è definito dalla costante IFC_CORBKC_DEFAULT_ROTATION_SPEED.
11. **raisingSpeed** - Parametro opzionale che consente di regolare la velocità di avvicinamento (allontanamento) al piano di riferimento. Il suo valore di default è definito dalla costante IFC_CORBKC_DEFAULT_RAISING_SPEED.

[8] Inteso come tipo di dato del linguaggio S3D.

12. **zoomSpeed** - Parametro opzionale che consente di regolare la velocità di avvicinamento (allontanamento) all'asse del cilindro. Il suo valore di default è definito dalla costante IFC_CORBKC_DEFAULT_ZOOM_SPEED.

Questi sono anche i parametri richiesti dal metodo *CreateInteractionModality.*

Mouse controller

È implementato dalla classe *IFCylindricalOrbitingMouseController*. L'interazione avviene tramite drag, in maniera del tutto analoga al mouse controller per la metafora *IFOrbiting*. L'unica differenza significativa rispetto a quest'ultimo consiste nella possibilità di assegnare, per la variazione dell'altezza o della distanza, lo stesso pulsante usato per ruotare attorno al cilindro. Infatti, quando l'utente inizia il drag, il controller verifica la direzione nella quale viene spostato il cursore. Se prevale la componente orizzontale si assume che l'utente voglia ruotare attorno al cilindro mentre, se prevale quella verticale, viene attivata l'altra modalità.

I principali parametri del costruttore (da usare anche come argomenti di *CreateInteractionModality*) sono:

4. **orbitButton** - Il pulsante usato per ruotare attorno all'asse del cilindro.
5. **raiseButton** - Il pulsante usato per variare l'altezza rispetto al piano di riferimento.
6. **zoomButton** - Il pulsante usato per avvicinare/allontanare la camera dall'asse del cilindro.
7. **useWheelForZoom** - Flag booleano per abilitare l'uso della "rotella" del mouse per effettuare lo zoom.
8. **zoomWheelSpeed** - Parametro opzionale che consente di regolare la sensibilità della rotella del mouse. Il suo valore di default è definito dalla costante IFC_CORBMC_DEFAULT_WHEEL_SENSIBILITY.
9. **rotationSensibility** - Parametro opzionale che consente di regolare velocità di rotazione. Il suo valore di default è definito dalla costante IFC_CORBMC_DEFAULT_ROTATION_SENSIBILITY.
10. **zoomSensibility** - Parametro opzionale che consente di regolare la velocità di avvicinamento (allontanamento) all'asse del cilindro. Il suo valore di default è definito dalla costante IFC_CORBMC_DEFAULT_ZOOM_SENSIBILITY.

11. **raisingSensibility** - Parametro opzionale che consente di regolare la velocità con la quale la camera si alza o si abbassa rispetto al piano di riferimento. Il suo valore di default è definito dalla costante IFC_CORBMC_DEFAULT_RAISING_-SENSIBILITY.

Touchscreen controller

È implementato dalla classe *IFCylindricalOrbitingTouchController*. Suddivide la finestra in due aree: l'area principale ed un margine attivo. Draggando lungo il margine si modifica la distanza tra la camera e l'asse del cilindro. Le azioni intraprese in seguito ad un drag nell'area principale dipendo invece dalla direzione di spostamento del cursore (in modo analogo a quanto avviene con il mouse): se prevale la componente orizzontale viene attivato l'orbiting, in caso contrario si agisce sull'altezza.

I parametri di interesse del controller sono:

4. **activeMargin** - Consente di scegliere dove collocare il margine attivo. Accetta uno dei seguenti valori: IFC_CORBTC_NO_ZOOM_ACTIVE_MARGIN, IFC_CORBTC_-ZOOM_ACTIVE_MARGIN_TOP, IFC_CORBTC_ZOOM_ACTIVE_MARGIN_-BOTTOM, IFC_CORBTC_ZOOM_ACTIVE_MARGIN_LEFT, IFC_CORBTC_-ZOOM_ACTIVE_MARGIN_RIGHT.

5. **marginOffset** - La distanza tra il margine attivo e l'angolo della finestra, espressa in pixel.

6. **marginDepth** - La larghezza del margine attivo.

7. **rotationSensibility** - Parametro opzionale che consente di regolare velocità di rotazione. Il suo valore di default è definito dalla costante IFC_CORBTC_DEFAULT_-ROTATION_SENSIBILITY.

8. **zoomSensibility** - Parametro opzionale che consente di regolare la velocità di avvicinamento (allontanamento) all'asse del cilindro. Il suo valore di default è definito dalla costante IFC_CORBTC_DEFAULT_ZOOM_SENSIBILITY.

9. **raisingSensibility** - Parametro opzionale che consente di regolare la velocità con la quale la camera si alza o si abbassa rispetto al piano di riferimento. Il suo valore di default è definito dalla costante IFC_CORBTC_DEFAULT_RAISING_-SENSIBILITY.

Kinect controller

La classe *IFCylindricalOrbitingKinectController* consente di controllare la metafora *Cylindrical Orbiting* per mezzo del Kinect. L'interazione è molto simile a quella implementata per l'Orbiting e viene illustrata in figura 5.9. Anche gli argomenti di questo controller sono gli stessi di *IFOrbitingKinectController*, salvo l'aggiunta di un ulteriore parametro (**raisingSensibility**) per regolare la sensibilità con la quale la camera si sposta verticalmente rispetto al piano di riferimento.

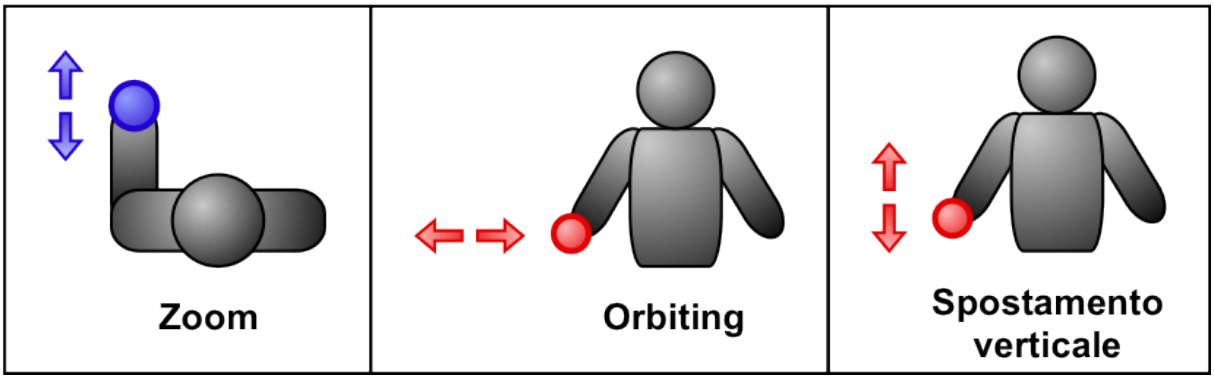


Figura 5.9: Uso del Microsoft Kinect per controllare la metafora *Cylindrical Orbiting*

5.5 La classe IFFlyingVehicle

Questa classe fornisce l'implementazione della metafora *Flying Vehicle*, vista nel paragrafo 1.5.1. Concettualmente, la camera è posta su di un aeromobile con capacità di manovra simili a quelle di un elicottero. È la metafora che, tra tutte, consente la maggiore libertà di navigazione grazie ai suoi 5/6 gradi di libertà (a seconda delle implementazioni). La classe *IFFlyingVehicle* fornisce cinque gradi di libertà: non è infatti possibile ruotare attorno all'asse longitudinale del veicolo (rollio). Tuttavia questo vincolo non compromette minimamente l'usabilità della metafora, in quanto il sesto grado di libertà risulta quasi sempre inutile e molto spesso tende a disorientare l'utente.

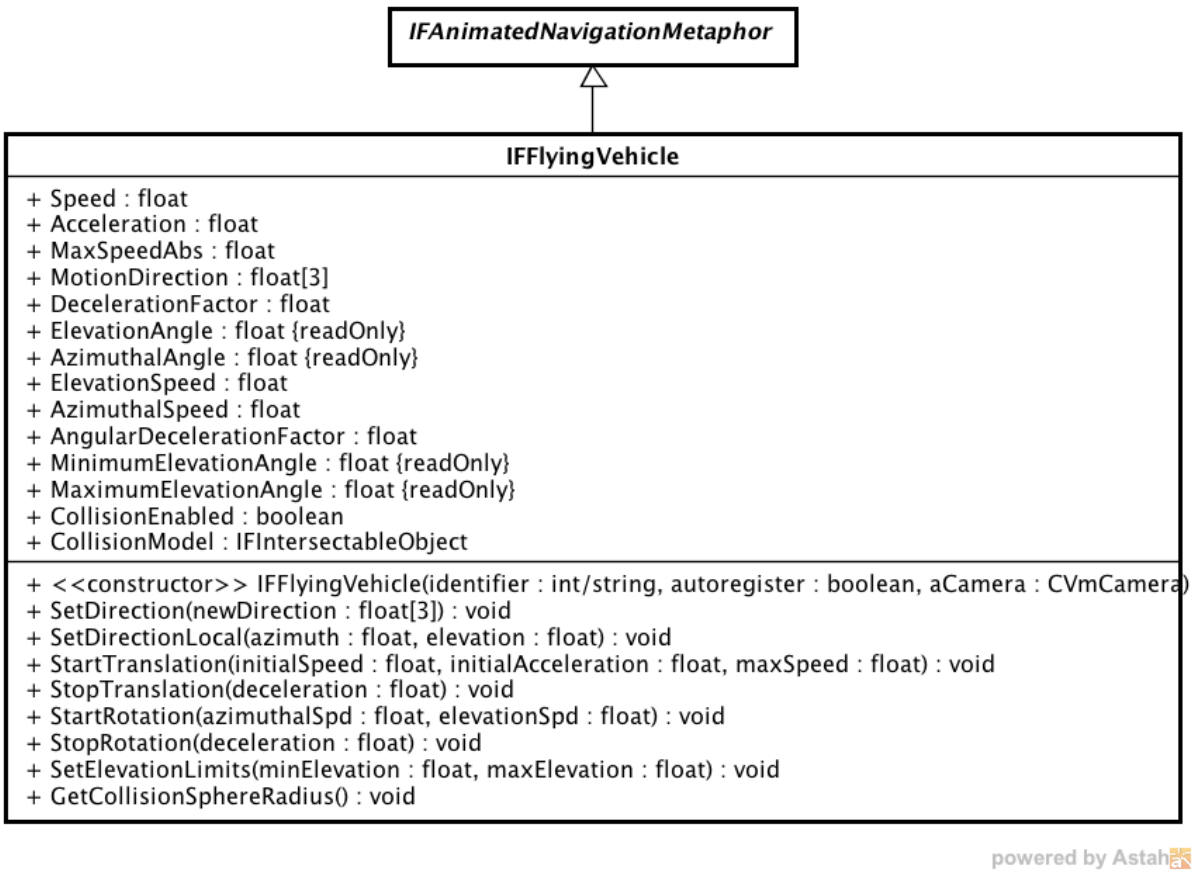


Figura 5.10: La classe *IFFlyingVehicle*

Posizione e direzione della camera

L'orientamento della camera è espresso in un sistema di riferimento sferico, quindi attraverso un angolo azimutale ed uno di elevazione, memorizzati rispettivamente nei campi **AzimuthalAngle** e **ElevationAngle**. A differenza delle metafore precedenti, però, l'origine del sistema di riferimento coincide con la posizione della camera. La direzione può essere impostata per mezzo dei metodi **SetDirection** o **SetDirectionLocal**, a seconda che si preferisca usare il sistema di riferimento sferico locale o quello cartesiano globale. Come sempre è presente il metodo **GetDirection** per ottenere la direzione in coordinate globali. L'elevazione non può mai superare gli 85^{o} in valore assoluto. Questo limite è stato scelto per evitare inclinazioni troppo ripide (se non addirittura inverse) che risultano difficili da controllare per l'utente. È comunque possibile imporre dei limiti più restrittivi grazie al metodo **SetElevationLimits**.

La posizione della camera è invece sempre espressa nel sistema di riferimento cartesiano globale, quindi non sono necessari né il metodo **SetPositionLocal**, né i campi per la memorizzazione delle coordinate locali. La gestione della posizione avviene quindi attraverso i metodi **Get**/**SetPosition** dichiarati da *IFNavigationMetaphor*.

Velocità e accelerazione

Salvo la fase di inizializzazione ed in corrispondenza dei cambi di modalità, è raro (oltre che sconsigliato) impostare direttamente la posizione della camera. In condizioni normali si dovrebbe invece regolare parametri come velocità e accelerazione, lasciando che sia la metafora a ricalcolare la posizione della camera ad ogni frame. La classe *IFFlyingVehicle* permette di far attraverso i seguenti campi:

Speed - La velocità di spostamento del veicolo, espressa come unità per frame.

Acceleration - L'accelerazione del veicolo. Ad ogni frame il valore contenuto in questo campo viene aggiunto alla velocità corrente.

MaxSpeedAbs - La massima velocità raggiungibile dal veicolo, in valore assoluto. Questo campo è presente in ogni metafora che supporta l'accelerazione, al fine di evitare che la velocità cresca indefinitamente.

MotionDirection - Vettore che esprime la direzione di spostamento. Il suo valore di default è NULL, ad indicare che viene usata come direzione di spostamento la direzione di vista. Quando invece si vuole muovere il veicolo in una direzione ben precisa e indipendente da quella di vista, basta assegnare tale direzione a questo campo (ricordandosi di normalizzarla).

DecelerationFactor - Il fattore di decelerazione da applicare ad ogni frame al campo *Speed*.

Come sempre è presente una coppia di metodi, **Start**/**StopTraslation** che semplifica la gestione di questi parametri, evitando al programmatore di settarli direttamente.

In analogia alle metafore precedenti, si può agire sull'orientamento del veicolo attraverso i campi **ElevationSpeed**, **AzimuthalSpeed** e **AngularDecelerationFactor**, ed i metodi di utilità **Start**/**StopRotation**.

Collision detection

Una delle funzionalità più interessanti di *IFFlyingVehicle* è la *collision detection*, ovvero la capacità di rilevare ed evitare la compenetrazione tra la camera e gli altri elementi della scena. Questa funzionalità è stata implementata sfruttando gli algoritmi e le strutture dati definiti nel package "*Intersection*", che vedremo nel capitolo 8. È bene precisare che, per ragioni di efficienza, in genere si ricorre a due modelli distinti per una stessa scena: uno è dedicato alla visualizzazione, l'altro ai test di collisione ed intersezione. La differenza tra i due consiste nel fatto che il primo è costituito quasi esclusivamente da mesh triangolari

e fornisce una rappresentazione accurata della scena da disegnare, mentre il secondo si propone di essere solo una approssimazione "grezza" della stessa e può essere costituito da mesh con un basso numero di poligoni e/o da apposite strutture dati che rappresentano vari tipi di primitive geometriche.

Il modo più semplice per implementare la collision detection consiste nel tracciare, ad ogni frame e partendo dalla posizione della corrente della camera, un segmento avente modulo e direzione pari a quelli del vettore velocità, e verificare se tale segmento interseca un qualche elemento della scena. In assenza di intersezioni si sposta la camera nel punto di destinazione, in caso contrario si procede come segue. Sia P la posizione corrente della camera, $\vec{d}$ la direzione di spostamento e C il punto di intersezione tra il segmento ed un elemento della scena. La nuova posizione della camera P' deve giacere sul segmento $\overline{PC}$ ad una distanza δ da C:

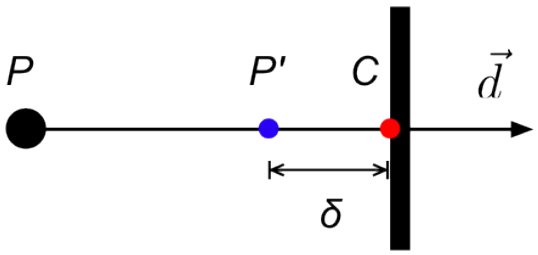


Figura 5.11: Collision detection mediante un segmento.

$$P' = P + (||C - P|| - \delta) \cdot \vec{d}$$

La scelta del valore di δ è lasciata al programmatore, purché sia maggiore della distanza tra la camera ed il near plane (il motivo di ciò sarà chiarito tra poco).

Questo approccio, sebbene semplice, produce però degli effetti grafici indesiderati. Supponiamo ad esempio che nella scena sia presente una parete e di posizionare la camera in prossimità di esso. Supponiamo inoltre di voler spostare la camera parallelamente alla parete in modo da non rilevare collisioni, come mostrato in figura 5.12 (a). Secondo le aspettative la scena dovrebbe venir visualizzata come mostrato in figura 5.12 (b). Tuttavia, se il near plane interseca uno o più elementi della scena (come nel caso del parallelepipedo usato per rappresentare la parete e del piano che modella il terreno) ed in conseguenza ai processi di *clipping* e *backface culling*, questi elementi appariranno come sezionati e perderanno la loro apparenza volumetrica. L'effetto ottenuto è quello mostrato in figura 5.12 (c).

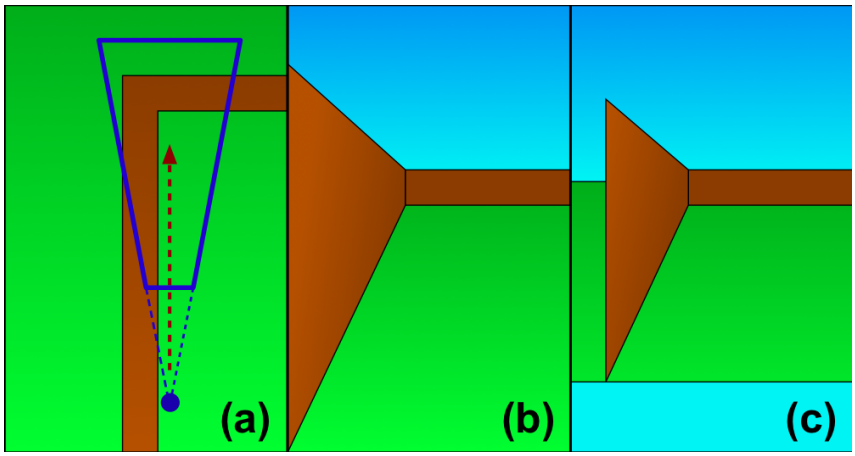


Figura 5.12: Intersezione tra il *near plane* e gli elementi della scena.

Una possibile soluzione a questo problema consiste nel far sì che la camera mantenga sempre una distanza opportuna da ogni elemento della scena, indipendentemente dalla direzione di spostamento. Questo equivale al racchiudere la camera all'interno di una sfera invisibile centrata su di essa e di raggio tale da inglobare completamente il *near plane*. La procedura di rilevazione delle collisioni può quindi essere modificata come segue:

- Ad ogni frame si applica il vettore velocità alla posizione corrente della camera, ottenendo un punto di destinazione D.
- Si imposta D come centro della sfera e se ne verifica l'intersezione con gli altri elementi della scena.
- Se non viene rilevata alcuna collisione si può posizionare la camera in D, altrimenti si mantiene la posizione corrente.

Anche questo algoritmo non è però sufficiente ad evitare tutte le inconsistenze, che diventano sempre più frequenti ed evidenti al crescere della velocità. Si consideri ad esempio la situazione mostrata in figura 5.13. Poiché l'intersezione viene testata solo nel punto di destinazione (anziché lungo tutto il tragitto), può capitare di oltrepassare completamente un oggetto contro il quale la camera avrebbe dovuto invece collidere. Questo fenomeno è noto come *tunneling* [11].

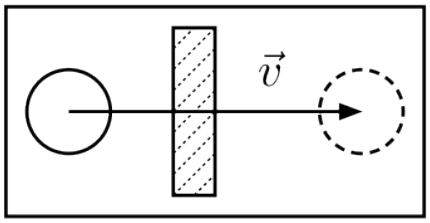


Figura 5.13: Tunneling

La soluzione adottata nel framework consiste nel verificare la presenza di collisioni lungo tutto il percorso della sfera. Poiché il movimento effettuato dalla sfera in un singolo frame è di tipo rettilineo uniforme, è semplice esprimere questo moto in funzione di un ipotetico tempo di percorrenza t :

$$\begin{cases} C(t) = C_0 + t \cdot \vec{v} \\ \qquad t \in [0, 1] \end{cases}$$

dove t rappresenta il parametro tempo (compreso tra 0 e 1), C_0 il centro della sfera prima dello spostamento, $\vec{v}$ il vettore velocità e $C(t)$ il centro della sfera lungo li percorso.

Il problema della *collision detection* tra una sfera in movimento ed una superficie si traduce quindi nel determinare il tempo t_c nel quale la distanza tra centro della sfera e la superficie considerata è pari al raggio della sfera stessa, come illustrato in figura 5.14. Questa tecnica è nota come "*sweep test*" [12].

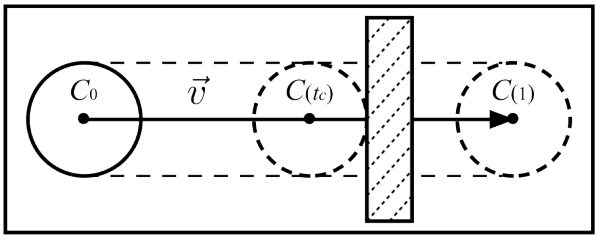


Figura 5.14: Collision detection mediante "*sweep test*".

Si noti che l'uso di questo approccio nelle applicazioni real-time è possibile solo in presenza di oggetti e tipi di moto semplici. In presenza di casi più complessi si deve perciò ricorrere a tecniche più sofisticate, come quelle descritte in [11].

Una volta individuata una collisione non ci si può limitare ad interrompere bruscamente il movimento. Invece la sfera (e quindi la camera) dovrebbe "scivolare" lungo un la superficie con la quale è entrata in contatto. Più precisamente, la sfera si deve muovere lungo il piano tangente alla sfera nel punto di collisione, detto "*sliding plane*" [12]. Se la sfera collide con una superficie piana, tale piano giacerà sulla superficie stessa. Ci sono però dei casi in cui questo non è possibile, come in presenza di superfici curve oppure quando la collisione avviene in corrispondenza di un bordo o di un vertice.

Sia C il centro della sfera al momento della collisione e P il punto di contatto tra la sfera e l'elemento della scena con cui essa collide. Supponiamo inoltre che in assenza di collisione la sfera avrebbe dovuto spostarsi nel punto di destinazione D.

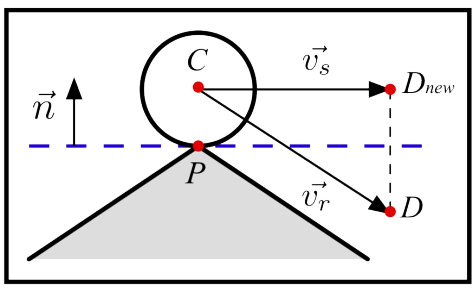


Figura 5.15: Sliding plane

Lo *sliding plane* può essere definito per mezzo di un punto che giace su di esso e di una normale. Il punto sarà ovviamente quello di contatto (P), mentre la normale si ottiene normalizzando il vettore che va da P a C:

$$\vec{n} = \frac{(C - P)}{||C - P||}$$

Poiché la sfera doveva raggiungere il punto D, si può immaginare che al momento del contatto essa abbia una velocità "residua" $\vec{v_r} = (D - C)$. La velocità $\vec{v_s}$ con la quale la sfera "scivola" lungo il piano si ottiene quindi proiettando $\vec{v_r}$ sullo sliding plane:

$$\vec{v_s} = \vec{v_r} - \vec{n} \cdot (\vec{v_r} \cdot \vec{n})$$

Non resta che calcolare la nuova destinazione D_{new}:

$$D_{new} = C + \vec{v_s}$$

Tornando alla classe *IFFlyingVehicle*, essa espone due campi riguardanti la funzionalità di *collision detection*:

CollisionEnabled - Flag booleano per abilitare o disabilitare la collision detection.

CollisionModel - In esso deve essere memorizzato un riferimento all'oggetto geometrico (o ad un insieme di oggetti geometrici) sui quale eseguire il test di collisione. Tale oggetto deve implementare l'interfaccia *IFIntersectableObject* definita nel package "*Intersection*" (si veda il capitolo 8).

Si ricorda infine che la sfera invisibile centrata sulla camera ha un raggio tale da inglobare completamente il near plane. Il valore del raggio dipenderà quindi dal rapporto tra larghezza e altezza della finestra e può cambiare durante l'esecuzione del programma. Questi aspetti vanno tenuti in considerazione in fase di progettazione e realizzazione della scena.

Animazioni

L'unico tipo di animazione supportato da *IFFlyingVehicle* è IFC_FV_ANIMATE_DIRECTION, attraverso la quale è possibile regolare l'orientamento della camera. I parametri per questa animazione sono:

newDirection - La nuova direzione della camera, espressa nel sistema di riferimento globale.

angularVelocity - Parametro opzionale che consente di regolare la velocità di rotazione. Il suo valore di default è definito dalla costante IFC_FV_ANIM_DIR_DEFAULT_ANGULAR_VELOCITY.

5.5.1 I controller di IFFlyingVehicle

Keyboard controller

È implementato dalla classe *IFFlyingKeyboardController*. I principali parametri che ne influenzano il comportamento sono:

- **initSpeed**- La velocità iniziale del veicolo, espressa in unità per frame.
- **acceleration** - L'accelerazione del veicolo, espressa come variazione di velocità per frame.
- **maxSpeed** - La massima velocità raggiungibile, espressa in unità per frame.
- **moveForwardButton** , **moveBackwardButton** , **moveRightButton** , **moveLeftButton** - I pulsanti della tastiera usati per muovere il veicolo.
- **lookUpButton** , **lookDownButton** , **lookRightButton** , **lookLeftButton** - I pulsanti della tastiera usati per modificare l'orientamento del veicolo.
- **rotationSpeed** - Parametro opzionale che consente di regolare la velocità di rotazione della camera. Il suo valore di default è definito dalla costante IFC_FKC_DEFAULT_ROTATION_SPEED.

Mouse controller

Il controller del mouse viene implementato dalla classe *IFFlyingMouseController*. L'interazione è molto semplice e simile a quella realizzata dagli altri "mouse controller" visti sinora. In particolare l'utente può modificare l'orientamento del veicolo "draggando" con uno specifico tasto del mouse (configurabile). La pressione di un altro pulsante consente invece di muovere il veicolo con velocità costante lungo la direzione corrente di vista. Durante il modo è anche possibile modificare la direzione con un drag e senza la necessità di premere ulteriori pulsanti. È infine possibile configurare un pulsante per spostare il veicolo in direzione inversa a quella di vista.

I parametri di interesse di questo controller rispecchiano questo tipo di interazione utente:

- **initSpeed**- La velocità iniziale del veicolo, espressa in unità per frame.
- **acceleration** - L'accelerazione del veicolo, espressa come variazione di velocità per frame.
- **maxSpeed** - La massima velocità raggiungibile, espressa in unità per frame.

- **translationMouseButton** - Il pulsante per avviare il moto. È possibile, durante la pressione, spostare il mouse per modificare l'orientamento del veicolo.
- **inverseTranslationMouseButton** - Il pulsante che avvia un moto in direzione inversa a quella di vista. È possibile, durante la pressione, spostare il mouse per modificare l'orientamento del veicolo.
- **rotationMouseButton** - Il pulsante del mouse adibito alla modifica dell'orientamento del veicolo (quando non è in movimento).
- **rotationSensibility** - Parametro opzionale che consente di regolare la velocità di rotazione della camera. Il suo valore di default è definito dalla costante IFC_FMC_DEFAULT_ROTATION_SENSIBILITY.

Touchscreen controller

È implementato dalla classe *IFFlyingTouchController*. A causa delle limitate capacità di controllo messe a disposizione dal touchscreen (due gradi di libertà e l'equivalente di un solo pulsante), si è dovuto ricorrere ad una interazione basata su stati, illustrata in figura 5.16.

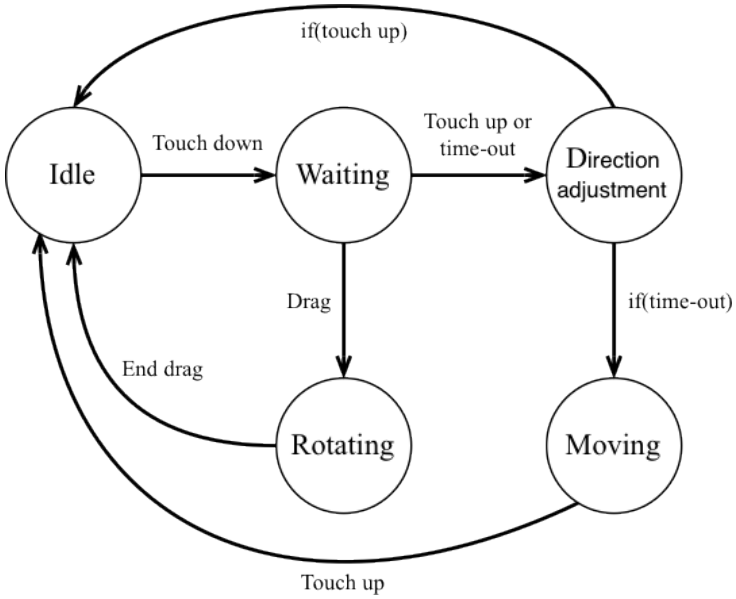


Figura 5.16:

L'interazione inizia dallo stato "*idle*". Quanto l'utente tocca lo schermo si passa allo stato "*waiting*", nel quale il controller resta in attesa di un'azione o dello scadere di un timer. L'utente ha ora tre possibilità: rilasciare il tocco, aspettare il time-out o iniziare un *drag*. Nel primo caso la camera viene ruotata (tramite una animazione) fino a centrare sullo schermo la porzione di scena corrispondente al punto nel quale l'utente ha eseguito il

tocco. In caso di "timeout", invece, dopo la fase di rotazione automatica la camera inizia a spostarsi. Il moto è di tipo uniformemente accelerato, in modo da massimizzare la manovrabilità del veicolo su tratte brevi, consentendo al contempo di percorrere grandi distanze in tempi brevi. Infine, se l'utente inizia un *drag* prima della scadenza del timer, a camera inizierà a ruotare seguendo il tocco dell'utente. Si noti che anche quando il veicolo è in movimento, l'utente può agire sull'orientamento della camera (e quindi sulla direzione di spostamento).

I principali parametri che influenzano il comportamento del controller sono:

- **initSpeed**- La velocità iniziale del veicolo, espressa in unità per frame.
- **acceleration** - L'accelerazione del veicolo, espressa come variazione di velocità per frame.
- **maxSpeed** - La massima velocità raggiungibile, espressa in unità per frame.
- **rotationSensibility** - Parametro opzionale che consente di regolare la velocità di rotazione della camera. Il suo valore di default è definito dalla costante IFC_FTC_-DEFAULT_ROTATION_SENSIBILITY.

Kinect controller

La classe *IFFlyingKinectController* fornisce una tecnica di interazione per il Kinect. Questo controller prevede che l'utente regoli la direzione di vista (e quindi di spostamento) attraverso un "*grab*" con la mano principale. Avvicinando o allontanando la mano sinistra dal corpo, invece, si regola la velocità del veicolo (che può anche essere negativa).

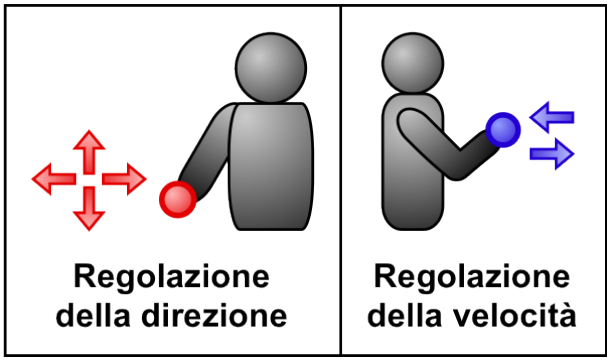


Figura 5.17: Uso del Microsoft Kinect per controllare la metafora *Flying vehicle*

I principali parametri del costruttore sono:

4. **enabled** - Flag opzionale che, se impostato a *false* o assente, fa sì che la modalità di default sia "disabilitata".

5. **speedSensibility** - Parametro opzionale che consente di regolare la velocità del veicolo. Questa è infatti calcolata come il prodotto tra il valore di questo parametro e la distanza tra la posizione corrente della mano ed il punto di inizio del "*grab*". Il suo valore di default è definito dalla costante IFC_FKMC_DEFAULT_SPEED_SENSIBILITY.

6. **rotationSensibility** - Parametro opzionale che consente di regolare la velocità di rotazione. Il suo valore di default è definito dalla costante IFC_FKMC_DEFAULT_ROTATION_SENSIBILITY.

I parametri del metodo *CreateInteractionModality* sono gli ultimi due argomenti del costruttore (se necessari, sono opzionali).

5.6 La classe IFWalking

La classe *IFWalking* fornisce l'implementazione della metafora *Walking*, presentata nel paragrafo 1.5.2. In questa metafora la camera si muove all'interno dell'ambiente virtuale con le stesse modalità di un essere umano, anche se può essere usata per simulare veicoli terrestri. Rispetto al *Flying vehicle*, quindi, la camera è vincolata al terreno.

L'interfaccia esposta da questa classe è molto simile a quella di *IFFlyingVehicle*, così come il suo utilizzo (*collision detection* compresa). In questo paragrafo ci limiteremo quindi ad esaminare le sole differenze significative.

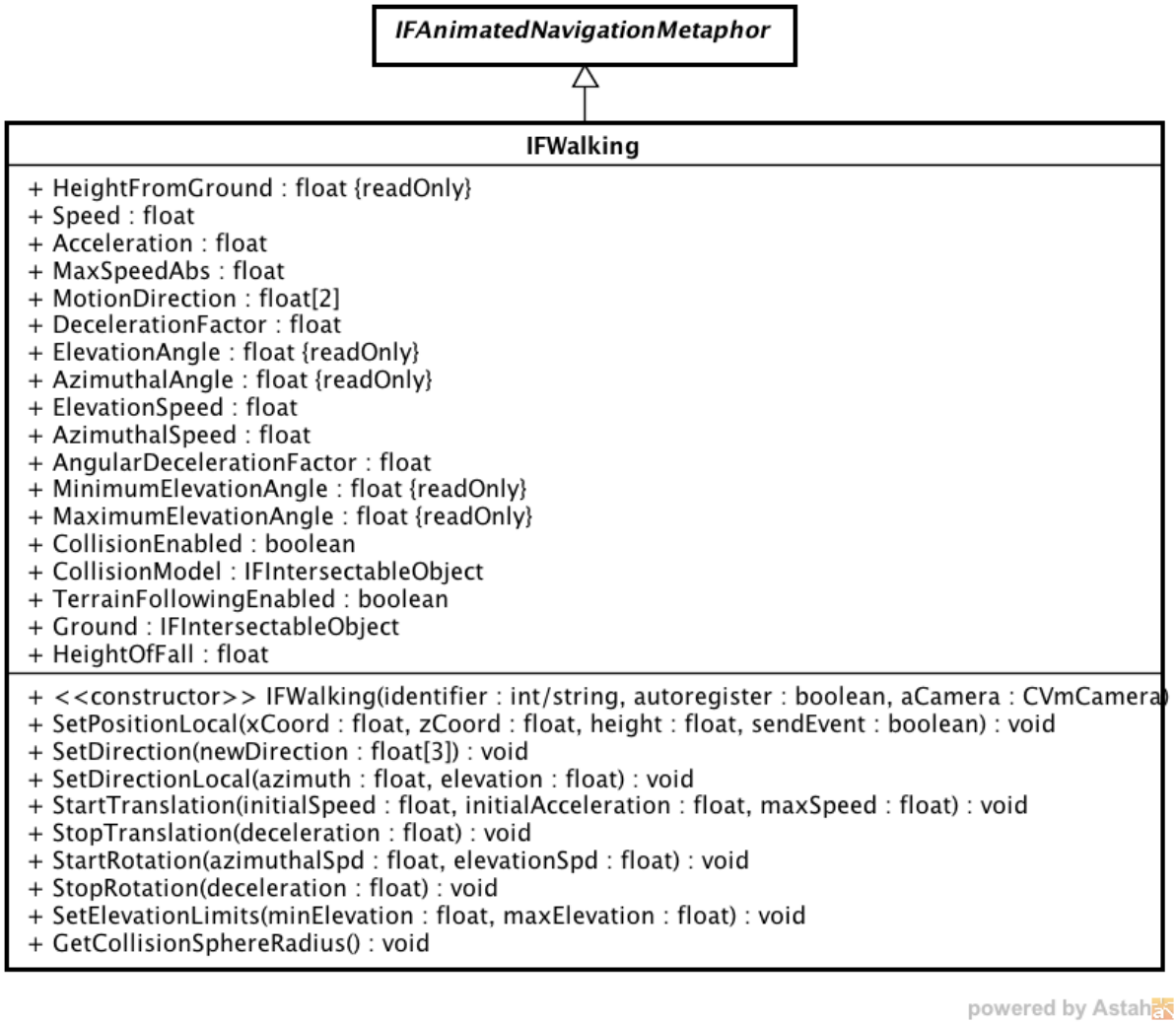


Figura 5.18: La classe *IFFlyingVehicle*

Terrain following

La principale differenza rispetto a *IFFlyingVehicle* consiste nel supporto al *terrain following*.

La posizione della camera viene espressa in un sistema di riferimento cartesiano locale e coincidente con quello globale. In questo sistema, tuttavia, la coordinata y esprime l'altezza della camera rispetto al terreno. Per tale ragione questa coordinata viene memorizzata in un apposito campo **(HeightFromGround)**. Le coordinate x e z si ottengono invece invocando il metodo **GetPosition**, che restituisce anche la coordinata y della camera espressa nel sistema di riferimento globale. La posizione della camera può essere impostata sia attraverso il metodo **SetPosition** sia con **SetPositionLocal**, a seconda se si preferisce esprimerla in coordinate globali o locali. È importante notare che nel primo caso, il campo *HeightFromGround* viene ricalcolato in funzione dell'elevazione del terreno in corrispondenza delle coordinate specificate.

I campi per abilitare la funzionalità di *terrain following* sono **TerrainFollowingEnabled** e **Ground**, aventi uno scopo del tutto analogo rispettivamente a **CollisionEnabled** e **CollisionModel**. Quando il *terrain following* è disabilitato (cioè quando *TerrainFollowingEnabled* è impostato a false oppure quando *Ground* contiene un riferimento nullo), si assume che il terreno sia costituito dal piano XZ. In questo caso il sistema di riferimento locale e quello globale coincidono.

Dal punto di vista implementativo, questa funzionalità viene realizzata proiettando nella scena una retta parallela all'asse Y e verificandone l'intersezione con l'oggetto memorizzato nel campo *Ground*. La coordinata y della camera viene quindi calcolata sommando il valore del campo *HeightFromGround* con la coordinata y del punto di intersezione.

Gravità

A seconda della morfologia del terreno è possibile incorrere in variazioni troppo repentine della posizione verticale della camera, a causa di zone troppo ripide o troppo scoscese. Per le prime si consiglia impedirne la "scalata" sfruttando il meccanismo della *collision detection*, mentre per ovviare alle seconde è stata implementata una simulazione della gravità. In particolare, quando nella procedura di *terrain following* si rileva un dislivello maggiore o uguale al valore contenuto nel campo **HeightOfFall**, la camera viene sottoposta ad una accelerazione costante verso il basso.

È bene precisare subito che l'implementazione data in questa classe non ha l'obiettivo di fornire una simulazione accurata dal punto di vista fisico, ma solo di risultare sufficientemente convincente e gradevole per l'utente, tenendo anche conto dell'ambito applicativo a cui è destinata. Nell'attuale implementazione, infatti, la gravità viene attivata in una modalità distinta dal resto del moto e non tiene quindi conto della velocità al momento della "caduta", ma solo della direzione di spostamento.

Per meglio chiarire i meccanismi che influenzano lo spostamento della camera, e come questi si combinano tra loro, di seguito viene descritto (ad alto livello) l'algoritmo usato per l'aggiornamento della posizione della camera:

- Ad ogni invocazione del metodo *Update* (generalmente ad ogni frame) si calcola la nuova destinazione della camera a partire dalla direzione di spostamento, dalla velocità e dall'accelerazione.
- Viene eseguito un test di collisione (come descritto a proposito della classe *IFFlyingVehicle*) per verificare che la camera non si avvicini troppo agli altri elementi della scena, e viene ricalcolato il punto di destinazione di conseguenza.
- Si calcola il punto di intersezione tra una retta verticale passante per il punto di destinazione e gli oggetti che costituiscono il modello del terreno, ottenendo così l'elevazione del terreno in quel punto[9].
- Se la differenza tra l'elevazione del terreno e la coordinata y corrente della camera è superiore al valore contenuto nel campo *HeightFromGround*, la camera viene spostata a destinazione, dopodiché si attiva la simulazione della gravità (che avrà effetto nelle successive invocazioni del metodo *Update*). In caso contrario la camera viene comunque spostata nel punto di destinazione, impostando però la coordinata y al valore ottenuto sommando l'elevazione del terreno con il valore del campo *HeightFromGround*.

Quanto alla simulazione della gravità, è importante notare che non basta applicare un'accelerazione costante verso il basso. Sono infatti possibili casi come quelli mostrati in figura 5.19 (a) dove, una volta terminata la "caduta", la camera si viene a trovare ad una distanza insufficiente dagli altri elementi della scena, riproponendo lo stesso problema illustrato in figura 5.12 (c). Per tale ragione occorre innanzi tutto trovare un punto idoneo in cui far terminare la caduta. La ricerca consiste in un ciclo nel quale, ad ogni iterazione, si effettua un test di intersezione **statica** tra la "sfera di collisione" ed il resto della scena[10]. In presenza di intersezione si prova a spostare la sfera seguendo la direzione di spostamento della metafora e di una quantità pari al raggio della sfera stessa (figura 5.19 (b)). Si ricalcola quindi l'elevazione del terreno in quel punto, si modifica la posizione verticale della sfera e si passa all'iterazione successiva, ripetendo il test. Il ciclo termina con l'individuazione di un punto di caduta idoneo oppure dopo un numero prestabilito di iterazioni. Per evitare un effetto del tipo "camminata nel vuoto", la direzione di caduta

[9] In caso di mancata intersezione si ricorre al piano XZ come modello del terreno, ottenendo così un'elevazione pari a 0.

[10] Più precisamente si verifica l'intersezione con i modelli del terreno e con quelli usati per la collision detection.

viene calcolata come la differenza tra il punto di destinazione e la nuova posizione corrente della camera (cioè quella ottenuta dopo la fase di collision detection), come mostrato in figura 5.19 (c). Da questo momento in poi, ad ogni invocazione del metodo *Update*, la camera si sposterà lungo la direzione di caduta con un'accelerazione costante, fino a quando non raggiungerà la giusta altezza dal terreno.

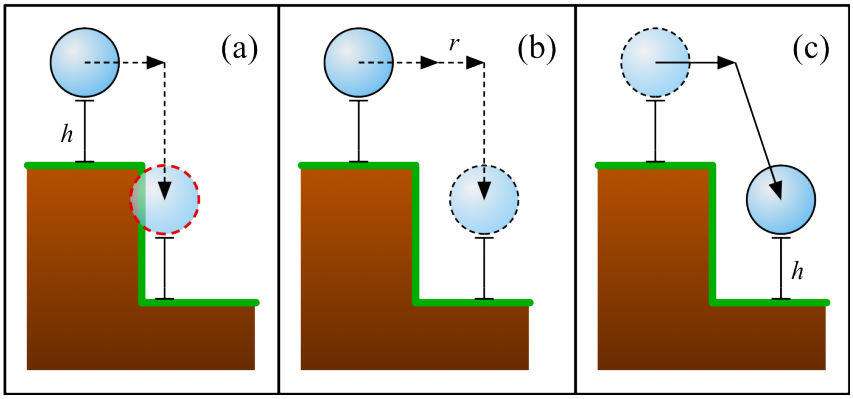


Figura 5.19: Algoritmo per la simulazione della gravità

Come anticipato, l'algoritmo usato per simulare il moto della camera non ha l'obiettivo di realizzare una simulazione fisicamente accurata, ma solo quello di risultare sufficientemente credibile per utente, tenendo anche conto dell'ambito applicativo previsto per il suo impiego (fruizione di contenuti culturali). Inoltre, sebbene questo algoritmo cerchi di evitare per quanto possibile il verificarsi di inconsistenze nel posizionamento della camera[11], queste possono ancora verificarsi se nella realizzazione della scena non si tiene conto delle caratteristiche e dei limiti dell'algoritmo stesso.

Altre differenze rispetto a *IFFlyingVehicle*

Il campo **MotionDirection** viene usato per specificare la direzione di spostamento, tuttavia tale direzione non viene espressa nel sistema di riferimento globale, bensì in un sistema di riferimento proprio. In particolare il vettore *MotionDirection* consta di due componenti: la prima coordinata rappresenta la proiezione della direzione di vista sul piano XZ, mentre la seconda è data dalla proiezione sullo stesso piano dell'asse X del sistema di riferimento locale alla camera. In sostanza, assegnando la prima coordinata ad 1 (-1) si ottiene uno spostamento in avanti (indietro), mentre assegnando la seconda ad 1 (-1) si ottiene uno spostamento verso destra (sinistra). Combinando le due (e normalizzando il vettore) si ottiene uno spostamento in diagonale.

Quanto alle animazioni, ed in analogia a *IFFlyingVehicle*, questa classe supporta la sola regolazione della direzione della camera, ottenuta specificando la costante IFC_WLK_-

[11] Ovvero posizionamenti che violano la semantica della metafora.

ANIMATE_DIRECTION come tipo di animazione. I parametri sono gli stessi richiesti da *IFFlyingVehicle*.

5.6.1 I controller di IFWalking

I controller per la classe *IFWalking* (implementati dalle classi *IFWalkingKeyboardController*, *IFWalkingKinectController*, *IFWalkingMouseController* e *IFWalkingTouchController*) sono del tutto analoghi a quelli del *flying vehicle*, sia dal punto di vista dell'interazione utente che per i parametri richiesti.

L'unica differenza significativa si ha nel caso del touchscreen. Infatti, analogamente a quanto avviene nei controller per touchscreen relativi alle metafore *IFOrbiting* e *IFCylindricalOrbiting*, l'area dello schermo viene divisa in due regioni: un'area principale ed una "regione attiva" situata nella parte inferiore della finestra. Il comportamento in entrambe le regioni è quello finora descritto, con l'unica differenza che agendo sulla "regione attiva" si ottiene uno spostamento in direzione inversa a quella di vista.

Figura 5.20: Area attiva definita da *IFWalkingTouchController*

Il motivo per cui questa regione attiva non è stata implementata nel controller del *Flying Vehicle* risiede nel fatto che, potendo questa metafora muoversi anche verticalmente, l'input immesso nella regione inferiore della finestra è in genere più significativo di quanto non lo sia per il *Walking* (in assenza di questa regione).

5.7 La classe IFSpeedCoupledFlying

La classe IFSpeedCoupledFlying fornisce un'implementazione della metafora *Speed-coupled Flying*, descritta nel paragrafo 1.5.3. Questa metafora combina una navigazione immersiva a bassa velocità molto simile al *Walking* con una navigazione ad alta velocità e a "volo d'uccello". L'idea è infatti quella di mettere in relazione la velocità del veicolo virtuale con la sua altezza rispetto al terreno e alla sua inclinazione.

In figura 5.21 viene presentata l'interfaccia esposta dalla classe. Come si può notare è molto simile a quella esposta da *IFWalking*, così come il suo utilizzo. Le poche differenze tra le due classi dipendono dalla modalità di volo implementata da *IFSpeedCoupledFlying* e dai vincoli che questa introduce. Infatti l'utente non può più controllare direttamente l'inclinazione della camera, essendo questa vincolata alla velocità. Per tale ragione non

sono più presenti i campi *ElevationAngle*, *ElevationSpeed* ecc... Sempre per lo stesso motivo non è disponibile il metodo *SetDirection* che consentiva di impostare arbitrariamente la direzione della camera. La sua controparte, *SetDirectionLocal*, permette ora di regolare il solo azimuth. Come per *IFWalking*, la posizione è espressa in un sistema di riferimento cartesiano locale e coincidente con quello globale, dove la coordinata y esprime l'altezza della camera rispetto al terreno. Tuttavia, in questo caso, il valore che assume la coordinata y nel sistema globale non dipende più solo dall'elevazione del terreno, ma anche dalla velocità della camera.

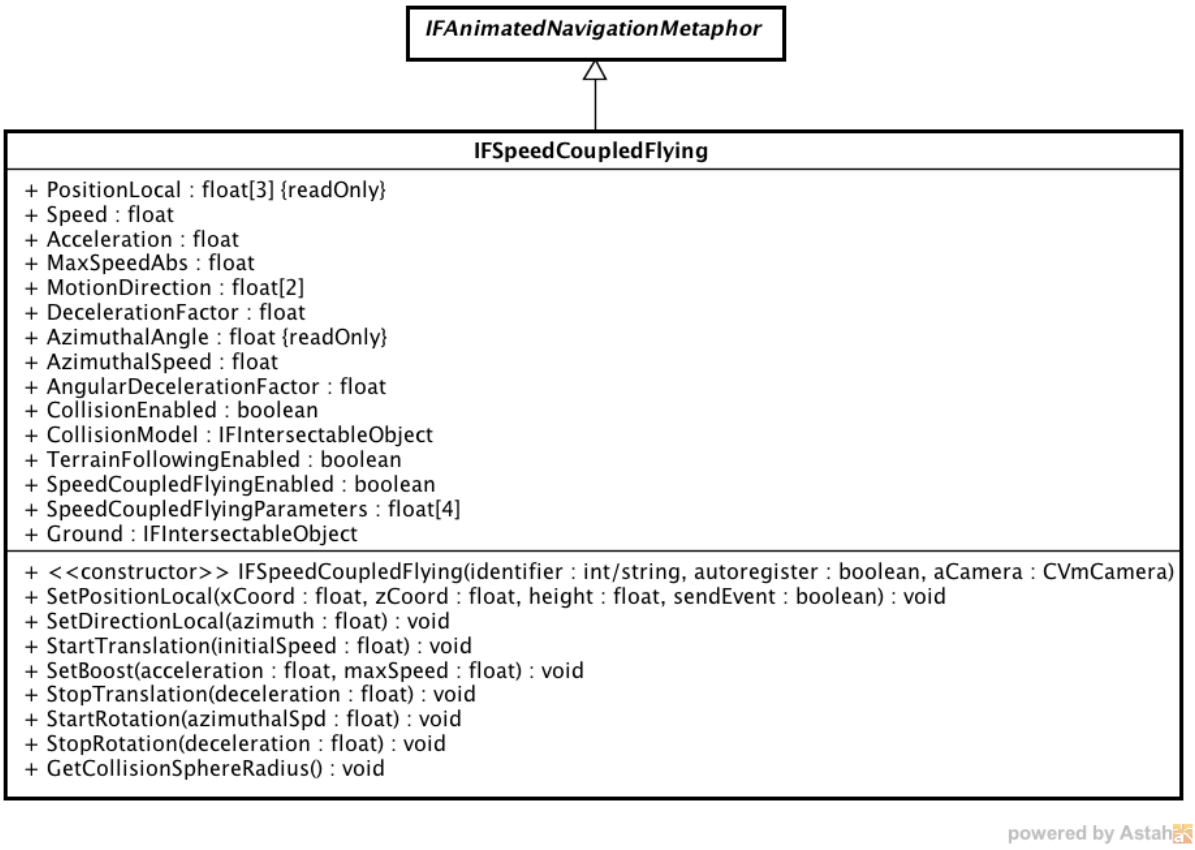


Figura 5.21: La classe *IFSpeedCoupledFlying*

La modalità di volo può essere abilitata o disabilitata per mezzo del campo booleano **SpeedCoupledFlyingEnabled**. I parametri che regolano il comportamento della metafora in questa modalità sono configurabili per mezzo del campo **SpeedCoupledFlyingParameters** e devono essere espressi come un vettore contenente:

speedThreshold - La velocità di soglia oltre la quale viene attivato il volo (espressa come unità per frame).

fullSpeed - La velocità alla quale si raggiunge la massima inclinazione della camera e la massima altezza rispetto al terreno. Questa velocità può non essere la massima velocità raggiungibile dal veicolo.

deltaHeight - Il massimo incremento di altezza dal suolo. Viene raggiunta quando la velocità è pari o superiore a *fullSpeed*.

maximumInclination - La massima inclinazione della camera, espressa in radianti. Viene raggiunta quando la velocità è pari o superiore a *fullSpeed*.

Ad ogni invocazione del metodo *Update*, la metafora verifica se la velocità ha superato la soglia. Se questo accade, altezza e inclinazione vengono calcolate per mezzo di una interpolazione lineare:

$$\begin{aligned} t &= max\left(\left(\frac{Speed - speedThreshold}{fullSpeed - speedThreshold}\right), 1\right) \\ Inclinazione &= maximumInclination \cdot t \\ PosizioneCamera.y &= PositionLocal.y + elevazioneTerreno + (deltaHeight \cdot t) \end{aligned}$$

In genere la navigazione a bassa velocità avviene con velocità costante, mentre un'eventuale accelerazione (con conseguente attivazione della modalità di volo) viene richiesta esplicitamente dall'utente, ad esempio premendo un pulsante apposito. Queste considerazioni si riflettono sul modo in cui è stato concepito il metodo **StartTraslation** per questa classe, che infatti richiede la sola velocità come parametro, mentre l'accelerazione viene invece impostata tramite un metodo distinto (**SetBoost**).

Quanto alle animazioni, anche questa classe supporta la sola regolazione della direzione della camera. Tuttavia, a causa dei vincoli imposti dalla metafora, è possibile regolare il solo azimuth.

5.7.1 I controller di IFSpeedCoupledFlying

Le tecniche di interazione implementate dai controller di *IFSpeedCoupledFlying* sono del tutto analoghe a quelle viste per *IFWalking*. Le uniche differenze significative sono le seguenti:

- Non è presente il controller per il touchscreen, in quanto questo dispositivo si è dimostrato palesemente inadatto al controllo della metafora.
- I controller per mouse e tastiera prevedono un tasto apposito per accelerare il veicolo ed attivare così la modalità di volo.
- Per quanto riguarda il Kinect, e in conseguenza di quanto spiegato fino ad ora, la mano secondaria non si limita a controllare la velocità, ma anche inclinazione e altezza dal suolo.

Le classi relative ai controller sono *IFSCFlyingKeyboardController*, *IFSCFlyingKinectController* e *IFSCFlyingMouseController*. I parametri da essi richiesti sono sostanzialmente gli stessi visti sino ad ora, e possono essere consultati nella documentazione.

5.8 La classe IFOverheadCrane

La metafora *IFOverheadCrane* ricorda, per il modo in cui viene gestita la camera, un carroponte. La camera è infatti virtualmente situata sul gancio di un carroponte e ne segue i movimenti. Dal punto di vista geometrico, questo si traduce nel vincolare gli spostamenti lungo un *piano di riferimento*, dando però la possibilità di variare la distanza verticale tra la camera ed il piano stesso. Questa metafora consente una visione dall'alto della scena ed è abbastanza utilizzata nella pratica.

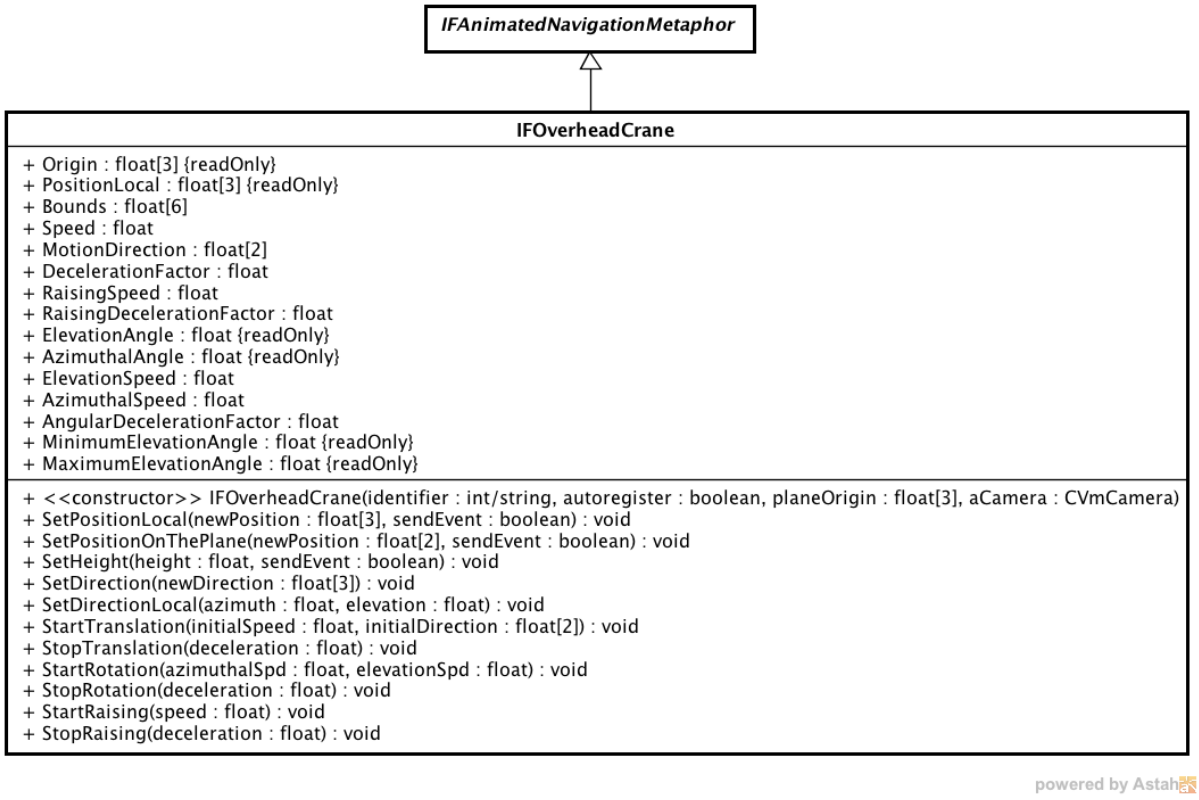


Figura 5.22: La classe *IFOverheadCrane*

IFOverheadCrane utilizza un sistema di riferimento cartesiano locale, ottenuto specificando il punto di origine e traslando il sistema di riferimento globale in tale punto. Il piano di riferimento passa per l'origine ed ha l'asse Y come normale. L'origine del sistema locale è uno dei parametri richiesti dal costruttore e viene memorizzata nel campo **Origin**. Le coordinate della camera sono invece memorizzate nel campo **PositionLocal**.

I metodi ed i campi di questa classe sono sostanzialmente gli stessi delle metafore viste sino ad ora, con alcune importanti differenze:

- È possibile modificare la sola posizione sul piano o la sola "altezza"[12] rispetto ad esso grazie ai metodi **SetPositionOnThePlane** e **SetHeight**.
- Velocità e direzione di spostamento (memorizzate nei campi **Speed** e **MotionDirection**) si riferiscono ai movimenti lungo il piano di riferimento. Alla velocità con la quale la camera si avvicina o si allontana verticalmente dal piano è destinato un apposito campo (**RaisingSpeed**), così come per il relativo fattore di decelerazione (**RaisingDecelerationFactor**). I metodi **Start**/**StopRaising** semplificano la gestione di questi campi.
- È possibile porre dei limiti alla posizione che la camera può assumere settando opportunamente il campo **Bounds**. Ad esso deve essere assegnato un array di sei elementi contenente le coordinate minime e massime consentite per ogni componente, espresse nel sistema di riferimento locale.

5.8.1 I controller di IFOverheadCrane

Le tecniche di interazione implementate dai controller di *IFOverheadCrane* sono simili a quelle viste per le altre metafore. Le uniche differenze significative sono le seguenti:

- Il controller del mouse prevede un apposito pulsante per variare l'altezza.
- Il controller del touchscreen sfrutta sia un margine attivo sia una regione attiva situata nella parte inferiore dello schermo. Il margine consente all'utente di spostare verticalmente la camera, mentre agendo sulla regione attiva si avvia uno spostamento lungo il piano in direzione inversa a quella di vista.
- L'interazione tramite Kinect avviene con le stesse modalità del *Walking*, con in più la possibilità di spostare verticalmente la camera muovendo verso l'alto o verso il basso la mano sinistra.

Le classi che implementano questi controller sono *IFCraneKeyboardController*, *IFCraneKinectController*, *IFCraneMouseController* e *IFCraneTouchController*.

[12] La distanza verticale dal piano di riferimento.

5.9 La classe IFAnimatedFlight

IFAnimatedFlight non rappresenta di per sé una metafora, ma costituisce la base di tutte le metafore basate su punti di interesse (esaminate nel paragrafo 1.5.7). Essa infatti implementa un insieme di animazioni per spostare automaticamente la camera verso un punto di interesse dato. È possibile scegliere tra varie funzioni per la velocità ed è anche possibile specificare la direzione che la camera deve assumere al termine dell'animazione.

Una volta avviata per mezzo del metodo **StartAnimation**, ogni chiamata del metodo *Update* produrrà un passo dell'animazione, avvicinando la camera al punto di destinazione e regolando gradualmente la sua direzione. Questa è l'unica tra le classi del framework che consente di concatenare più animazioni: ad ogni invocazione di *StartAnimation* viene creato un oggetto che implementa l'animazione, dopodiché tale oggetto viene inserito in una coda. Le animazioni in coda vengono quindi eseguite sequenzialmente, seguendo l'ordine di inserimento. È possibile conoscere il numero di animazioni in attesa di essere eseguite grazie al metodo **PendingAnimations**. **StopAnimation** (dichiarato da *IFAnimatedNavigationMetaphor*) consente invece di terminare l'intera animazione, interrompendo quella corrente e svuotando la coda. Se invece si desidera interrompere la sola animazione in esecuzione, passando così a quella successiva, basta invocare **StopCurrentAnimation**.

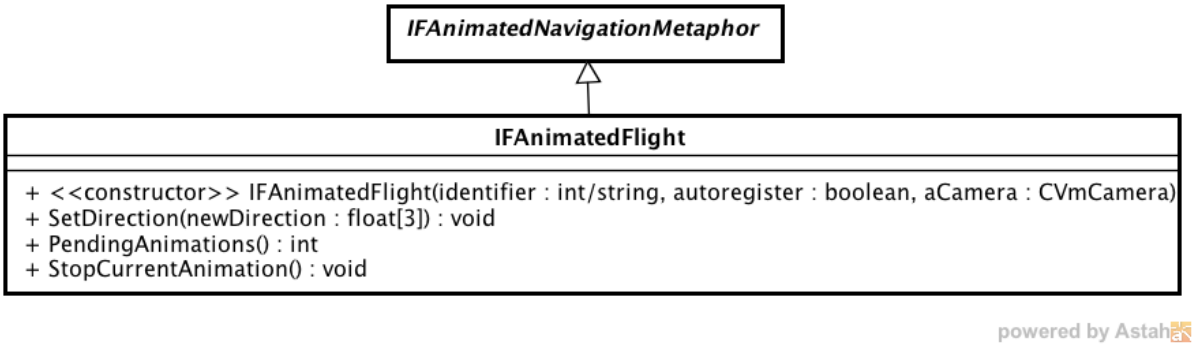


Figura 5.23: La classe *IFAnimatedFlight*

Considerando per il momento il solo posizionamento della camera, ad ogni passo dell'animazione la nuova posizione della camera viene calcolata come segue:

$$C_n = C_{n-1} + v_n \cdot \vec{d}$$

dove C_n rappresenta la posizione calcolata al passo n, v_n la velocità di spostamento (può cambiare ad ogni passo) e $\vec{d}$ è la direzione di spostamento.

Le animazioni implementate in questa classe si differenziano in base alla funzione velocità scelta. In particolare, $v\,(n)$ può essere la discretizzazione di una funzione costante, di una funzione esponenziale negativa oppure di una funzione a campana.

Funzione costante

È il caso più semplice, nel quale la velocità viene mantenuta costante per tutta l'animazione. Questo tipo di animazione è inadatta a percorrere lunghe distanze, a causa degli elevati tempi di percorrenza. Una soluzione potrebbe essere quella di impostare una velocità proporzionale alla distanza da percorrere. Tuttavia, se la velocità risultante è elevata, si rischia di ottenere partenze troppo repentine e arresti troppo bruschi.

La costante IFC_AF_NEGATIVE_EXPONENTIAL_SPEED_ANIMATION definisce l'identificatore associato a questa animazione. I suoi parametri sono rispettivamente:

- Il punto di interesse da raggiungere.
- La direzione che la camera deve assumere al termine dell'animazione.
- Il valore della velocità.

È anche possibile impostare il secondo parametro a NULL. In tal caso la direzione di vista coinciderà con quella di spostamento (per tutta la durata dell'animazione).

Funzione esponenziale

Una migliore soluzione viene presentata in [24], dove si propone di usare una *funzione esponenziale negativa*. Tale funzione può essere implementata efficientemente mediante la seguente successione ricorsiva:

$$v_n = k \cdot d_{n-1} = k \cdot (P - C_{n-1})$$

dove P è il punto di interesse da raggiungere, C_i è la posizione della camera al passo i-esimo e d_i è la distanza tra C_i e P.

Questa soluzione prevede quindi un'elevata velocità alla partenza, che poi decresce esponenzialmente col passare del tempo (e con l'avvicinarsi della destinazione). Così facendo è possibile percorrere grandi distanze in tempi accettabili[13] evitando al contempo un arresto brusco al termine dello spostamento. Tuttavia resta la possibilità di avere partenze troppo repentine, che possono disorientare l'utente.

L'identificatore associato a questa animazione è IFC_AF_NEGATIVE_EXPONENTIAL_SPEED_ANIMATION, mentre i parametri sono gli stessi del caso precedente.

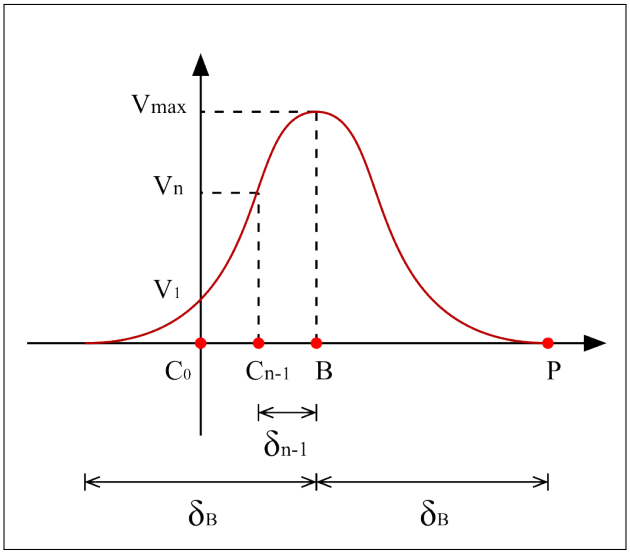
[13] Scegliendo una velocità iniziale proporzionale alla distanza da percorrere.

Funzione a campana

I problemi che si verificano con le soluzioni precedenti possono essere risolti ricorrendo ad una *funzione a campana*. Grazie la sua forma caratteristica, questa funzione consente velocità elevate pur garantendo una partenza ed un arrivo graduale.

Nell'implementazione, anziché essere valutata esplicitamente, la funzione viene approssimata dalla seguente successione ricorsiva:

$$v_n = V_{max} \cdot \left(1 - \frac{(B - C_{n-1}) \cdot (B - C_{n-1})}{\delta_B^2}\right) = V_{max} \cdot \left(1 - \frac{\delta_{n-1}^2}{\delta_B^2}\right)$$



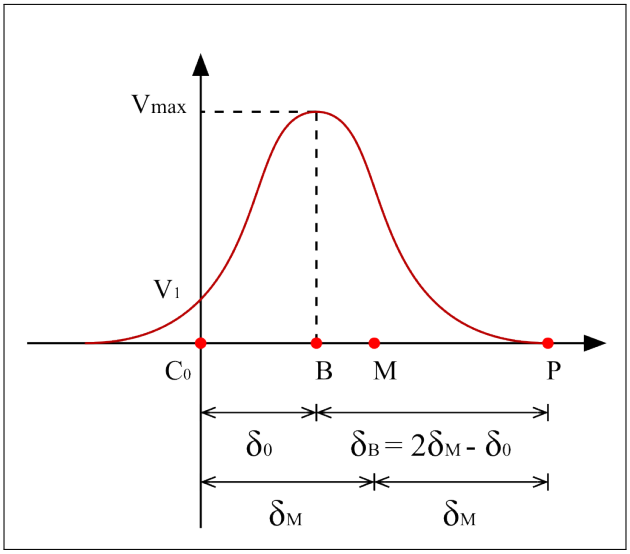
Dove:

- V_{max} rappresenta la velocità massima. Deve essere maggiore di zero.
- P è il punto di interesse (ovvero la destinazione).
- C_i è la posizione della camera al passo i-esimo dell'animazione (quindi C_0 è la sua posizione iniziale).
- v_i rappresenta la velocità con la quale spostare la camera al passo i-esimo. La velocità iniziale v_1 deve essere compresa tra zero e V_{max} (esclusi).
- B è il punto in corrispondenza del quale la funzione assume il suo valore massimo (V_{max}).
- La distanza tra B e P viene indicata con δ_B, mentre la distanza tra B e C_i viene indicata con δ_i.

Dati P, C_0, v_1 e V_{max} (sono i parametri dell'animazione), occorre innanzitutto ricavare δ_0 e di conseguenza B. Per far ciò partiamo dalle seguenti osservazioni.

Sia M il punto medio del segmento $\overline{C_0P}$ e δ_M la distanza tra M e gli estremi del segmento. Allora:

- B appartiene al segmento $\overline{C_0M}$ (estremi esclusi).
- La distanza δ_B può essere espressa come: $\delta_B = 2 \cdot \delta_M - \delta_0$.



Proviamo ora a valutare la successione al primo passo:

$$v_1 = V_{max} \cdot \left(1 - \frac{(B - C_0) \cdot (B - C_0)}{\delta_B^2}\right) = V_{max} \cdot \left(1 - \frac{\delta_0^2}{\delta_B^2}\right)$$

Sostituendo δ_B con $2 \cdot \delta_M - \delta_0$ si ottiene:

$$v_1 = V_{max} \cdot \left(1 - \frac{\delta_0^2}{(2 \cdot \delta_M - \delta_0)^2}\right) \Rightarrow 1 - \frac{v_1}{V_{max}} = \frac{\delta_0^2}{(2 \cdot \delta_M - \delta_0)^2}$$

Ponendo $K_v = 1 - {}^{v_1}/_{V_{max}}$ si ricava:

$$\delta_0^2 = K_v \cdot (2 \cdot \delta_M - \delta_0)^2$$

Risolvendo questa equazione di secondo grado si ottiene infine:

$$\begin{aligned} \delta_0 &= 2 \cdot \delta_M \cdot \left(\frac{\sqrt{K_v} - K_v}{1 - K_v}\right) \\ B &= C_0 + \delta_0 \cdot \left(\frac{P - C_0}{||P - C_0||}\right) \end{aligned}$$

Questo tipo di animazione può essere richiesta usando la costante IFC_AF_BELL_-SPEED_ANIMATION. I suoi parametri sono:

- Il punto di interesse da raggiungere.
- La direzione che la camera deve assumere al termine dell'animazione.
- La velocità iniziale.
- La velocità massima.

Ancora una volta è possibile impostare il secondo parametro a NULL, in modo da mantenere la camera orientata nella stessa direzione dello spostamento.

Animazioni relative alla direzione

Esiste anche un tipo di animazioni per la modificare la sola direzione della camera. In questo caso l'identificatore da usare è IFC_AF_DIRECTION_ANIMATION ed i parametri necessari sono la nuova direzione da assumere e la velocità di rotazione.

È interessante notare che quando viene richiesta un'animazione di spostamento, per prima cosa viene creata ed eseguita un'animazione di questo tipo, al fine di orientare automaticamente la camera nella direzione di spostamento.

Capitolo 6

Il package "SelectionMetaphors"

Il package *SelectionMetaphors* contiene le classi che implementano le metafore di selezione ed i relativi controller. Allo stato attuale è presente una metafora della famiglia "*virtual cursor*" ed una della famiglia "*virtual hand*". La prima può essere usata sia in ambienti desktop sia in quelli immersivi, mentre la seconda viene generalmente usata nei soli ambienti immersivi, dove si può disporre di un sistema di tracking. Nell'ambito di questa tesi è stato testato, come dispositivo di tracking, il Microsoft Kinect. Il framework, comunque, può essere facilmente esteso per supportare altri sistemi di tracking, grazie anche alla scelta di operare una netta distinzione (concettuale e realizzativa) tra metafore di interazione, tecniche di interazione e dispositivi di input.

6.1 La classe IFSelectionMetaphor

La classe astratta *IFSelectionMetaphor* rappresenta una generica metafora di selezione. Essa si basa sui concetti di **elemento selezionato** e di **candidato per la selezione**. Mentre il significato del primo è abbastanza ovvio, occorre spendere qualche parola sul secondo.

Prima di poter selezionare un oggetto è necessario eseguire un test per verificare se, in base alla semantica della specifica metafora, esiste un oggetto potenzialmente selezionabile. Se tale oggetto esiste, esso sarà il candidato corrente e potrà essere successivamente selezionato. La distinzione tra candidato ed oggetto selezionato permette all'utente di avere un feedback riguardo a quale oggetto andrà (eventualmente) a selezionare.

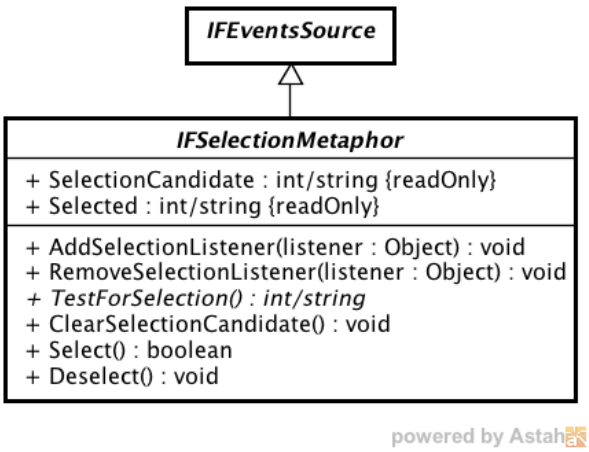


Figura 6.1: La classe *IFSelectionMetaphor*

Passiamo quindi ad esaminare la classe nel dettaglio. L'identificatore del candidato e dell'elemento selezionato sono memorizzati rispettivamente nei campi **SelectionCandidate** e **Selected**. Qualora uno dei due o entrambi non siano presenti, i relativi campi conterranno NULL.

Il test di selezione viene eseguito invocando il metodo **TestForSelection**, che restituisce l'identificatore dell'elemento potenzialmente selezionabile (se esiste) e lo memorizza nel campo *SelectionCandidate*. L'effettiva selezione avviene tramite il metodo **Select**. Si noti che se si chiama questo metodo in assenza di un candidato valido, questo non apporterà alcun cambiamento allo stato della metafora. Quindi, se quanto detto avviene in presenza di un oggetto precedentemente selezionato, esso rimarrà tale. Infine è possibile rimuovere il candidato e la selezione corrente rispettivamente attraverso i metodi **ClearSelectionCandidate** e **Deselect**.

Per fornire all'utente un feedback visivo (*highlighting*) riguardo al candidato per la selezione, è necessario invocare il metodo *TestForSelection* ad ogni frame, ed eseguire la selezione vera e propria solo su esplicita richiesta dell'utente. Se invece si preferisce fare a meno di questa funzionalità, i metodi *TestForSelection* e *Select* devono essere invocati in sequenza.

IFSelectionMetaphor genera due tipi eventi, **ObjectSelected** e **ObjectDeselected**, inviati rispettivamente quando un oggetto viene selezionato e quando viene deselezionato. I listener di questi eventi devono implementare, per la gestione degli stessi, i metodi **OnObjectSelected(objectId)** e **ObjectDeselected(objectId)**, il cui unico parametro è l'identificatore dell'oggetto selezionato/deselezionato. La registrazione ad entrambi gli eventi avviene grazie ai metodi **AddSelectionListener** e **RemoveSelectionListener**.

Infine vediamo alcune regole di base per la definizione di una metafora di selezione. Ogni sottoclasse concreta di *IFSelectionMetaphor* deve:

- Definire un costruttore che accetti come argomento (almeno) l'identificatore associato alla metafora, e inizializzare con esso il campo *Id* (ereditato da *IFEventsSource*).
- Implementare i metodi astratti *Register* e *Unregister* dichiarati dalla classe *IFEventsSource.*
- Consentire, per mezzo di appositi metodi o campi, di impostare un insieme di oggetti su cui eseguire il test di selezione.
- Implementare il metodo **TestForSelection**.

Le altre funzionalità descritte in questo paragrafo sono già implementate da *IFSelectionMetaphor.*

6.2 La classe IFRaySelection

La classe *IFRaySelection*, mostrata in figura 6.2, rappresenta una metafora di selezione basata su *ray-casting*. Il test selezione avviene quindi proiettando un raggio (ovvero una semiretta) nella scena e verificando l'intersezione tra lo stesso ed i modelli che compongono la scena. Se il raggio interseca più oggetti viene scelto quello più vicino all'origine del raggio.

È importante precisare che, sebbene un "*ray*" sia formalmente una semiretta, nella pratica è utile ricorrere a raggi di lunghezza finita, ovvero a **segmenti orientati** [27, 9]. Questa sarà pertanto la definizione di raggio usata in questa tesi.

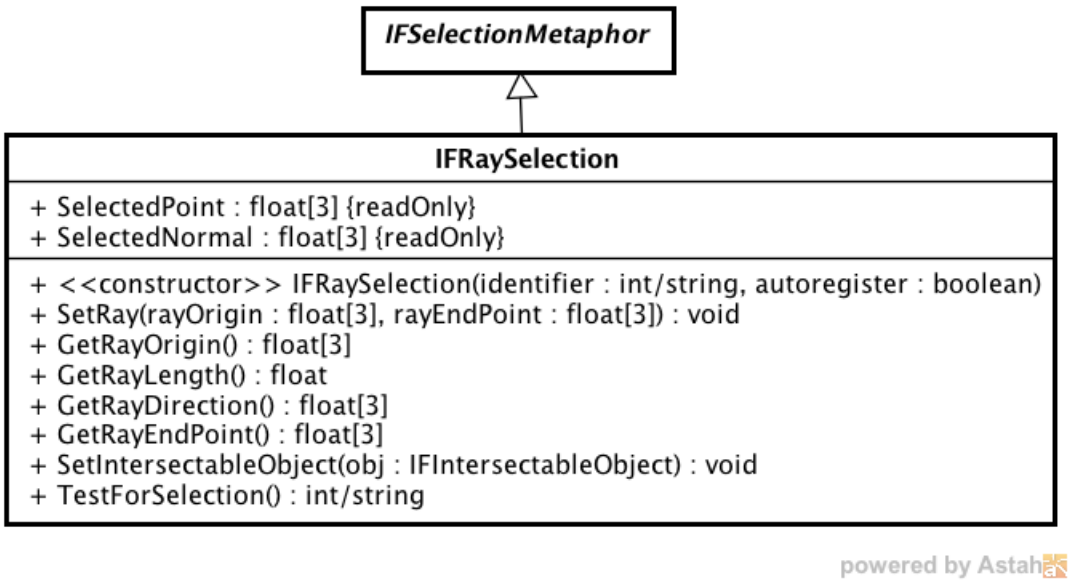


Figura 6.2: La classe *IFRaySelection*

La classe *IFRaySelection* mette a disposizione il metodo SetRay per impostare un raggio a partire dai suoi estremi. Sono inoltre presenti dei metodi "accessori" per ottenere le caratteristiche del raggio precedentemente impostato, quali l'origine, la lunghezza e la direzione.

Il test di selezione, implementato dal metodo **TestForSelection**, si basa sugli algoritmi e sulle classi definiti nel package "*Intersection*" e descritti nel capitolo 8. Di particolare importanza è la classe *IFIntersectableObject* che dichiara l'interfaccia implementata da ogni classe che supporta i test di intersezione. Tali classi possono modellare primitive geometriche, mesh di triangoli oppure un insieme di oggetti "intersecabili". Segue che *TestForSelection* in realtà si limita a delegare il test ad uno di questi oggetti, precedentemente impostato per mezzo del metodo **SetIntersectableObject**.

Una volta selezionato un oggetto (per mezzo del metodo **Select** ereditato da *IFSelectionMetaphor*), è possibile conoscere il punto di intersezione tra il raggio e l'oggetto selezionato e la normale alla superficie in quel punto grazie ai campi **SelectedPoint** e **SelectedNormal**.

6.2.1 I controller di IFRaySelection

Mouse controller

È implementato dalla classe *IFRaySelectionMouseController*. L'interazione avviene nella maniera più semplice possibile: "*click*" su di un oggetto per selezionarlo. A seconda di come viene configurato, è possibile far sì che l'elemento attualmente selezionato venga deselezionato quando fallisce il test di selezione, ovvero quando l'utente "clicca" su un punto della scena nel quale non sono presenti oggetti selezionabili.

I parametri che consentono di configurare il comportamento di questo controller[1] sono:

- **selectionButton** - Il bottone del mouse usato per selezionare un oggetto.
- **testEveryFrame** - Se impostato a *true* il test di selezione verrà eseguito ad ogni frame. L'identificatore dell'oggetto così individuato viene memorizzato nel campo **SelectionCandidate** (ereditato da *IFSelectionMetaphor*). È quindi possibile sfruttare questo campo per fornire un feedback visivo all'utente (*highlighting*). Se invece viene settato a *false*, il test di selezione viene svolto contestualmente alla selezione vera e propria.
- **deselectionButton** - Il bottone del mouse usato per rimuovere la selezione corrente. È possibile assegnare a questo parametro lo stesso valore usato per *selectionButton*.

[1] Ovvero i parametri più significativi del costruttore e richiesti dal metodo *CreateInteractionModality*.

Così facendo, se il click avviene sopra un oggetto selezionabile questo verrà selezionato, altrimenti verrà rimossa la selezione corrente (se presente). È un parametro opzionale quindi può essere omesso, settato a IFC_CTRL_UNASSIGNED oppure a NULL. In tal caso la deselezione sarà a carico di un altro oggetto definito dal programmatore.

Touchscreen controller

L'interazione via touchscreen è possibile grazie alla classe *IFRaySelectionTouchController*. Come si può immaginare l'utente può selezionare un oggetto con un semplice "tocco" su di esso. La deselezione avviene eseguendo il "tocco" in un punto della scena in corrispondenza del quale non sono presenti oggetti selezionabili.

Kinect controller

Viene implementato dalla classe *IFRaySelectionKinectController*. L'interazione avviene come mostrato in figura 6.3 ed è composta da tre fasi, da eseguire in sequenza:

1. *Attivazione del raggio*: il raggio viene generato solo quando la mano secondaria viene chiusa e fintanto che rimane in questo stato. È caldamente consigliato che l'utente, prima dell'attivazione, posizioni la mano (più o meno) all'altezza della spalla , come mostrato in figura. Così facendo, al momento della chiusura della mano il controller può approssimare la posizione delle spalle[2]. Si noti comunque che questa posizione di partenza è un consigliata in tutti i casi, in quanto consente al Kinect di rilevare correttamente la posizione delle mani e non limita la libertà di movimento ed il raggio d'azione dell'utente.

2. *Spostamento del raggio*: la direzione del raggio viene calcolata a partire dal vettore che congiunge la spalla alla mano principale. Segue che il "puntamento" avviene semplicemente orientando il braccio destro nella direzione desiderata. Il braccio deve essere mantenuto sufficientemente disteso e la mano deve restare aperta.

3. *Selezione/Deselezione*: si ottiene chiudendo la mano principale.

[2]L'applicazione *CaveNI* fornisce soltanto la posizione delle mani ed il baricentro dell'utente.

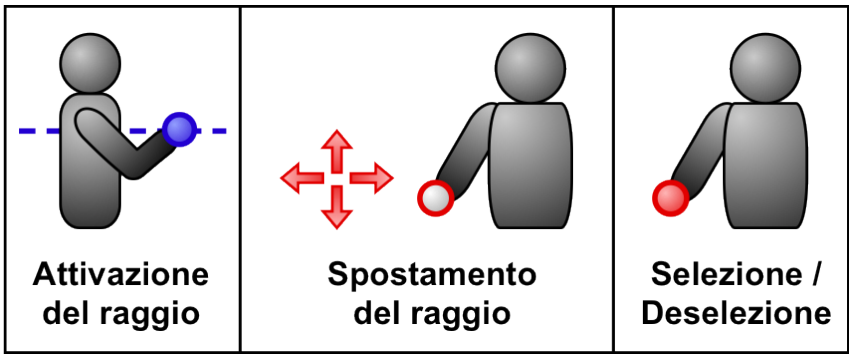


Figura 6.3: Uso del Microsoft Kinect per controllare la metafora *IFRaySelection*

L'unico parametro del costruttore degno di nota è **testEveryFrame**, avente lo stesso scopo del suo omonimo nella classe *IFRaySelectionMouseController*.

Infine questo controller espone il metodo **IsActive** che consente di conoscere lo stato di attivazione del raggio.

6.3 La classe IFCursorSelection

L'altra metafora di selezione presente nel framework appartiene alla famiglia "*virtual hand*", descritta nel paragrafo 1.6.1. Questa famiglia di metafore prevede l'inserimento nella scena di una rappresentazione virtuale della mano dell'utente, sotto forma di un cursore di qualche tipo. Il test selezione avviene testando l'intersezione tra il cursore e gli elementi della scena. Nel caso della classe *IFCursorSelection*, tale cursore è una sfera.

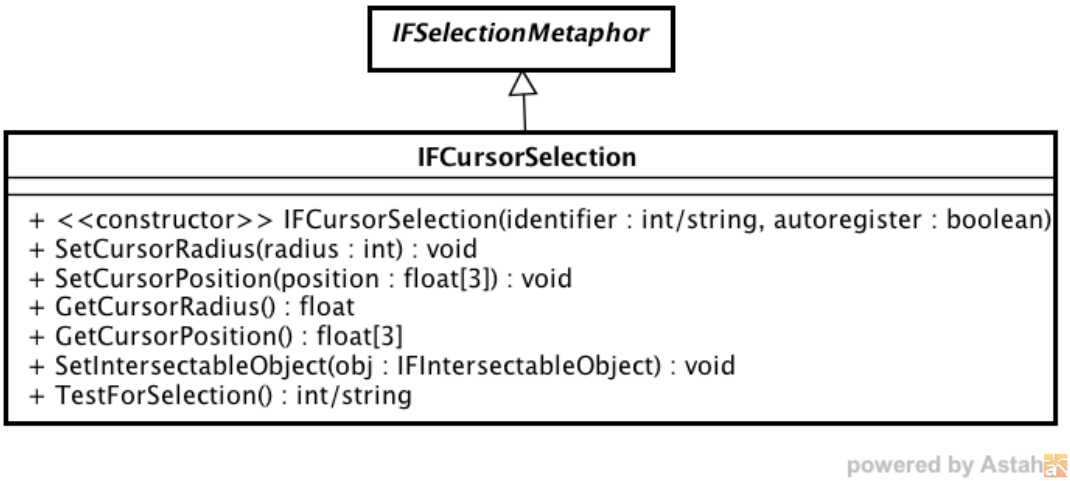


Figura 6.4: La classe *IFCursorSelection*

Posizione e raggio del cursore sono regolabili per mezzo dei metodi **SetCursorPosition** e **SetCursorRadius**, ed il loro valore è reperibile attraverso i corrispondenti metodi "accessori".

Come per la classe *IFRaySelection*, l'oggetto sul quale eseguire il test di intersezione viene settato dal metodo **SetIntersectableObject**.

6.3.1 I controller di IFCursorSelection

Mouse controller

È implementato dalla classe *IFCursorSelectionMouseController* ed è stato sviluppato principalmente per testare la classe *IFCursorSelection* e per fornire un esempio da seguire nello sviluppo di controller ad-hoc. Infatti le metafore della famiglia "*virtual hand*" trovano applicazione principalmente in ambienti immersivi e di rado vengono usate con i dispositivi tipici dei sistemi desktop.

Il cursore di selezione viene posizionato all'interno della scena in corrispondenza del cursore del mouse. La distanza dalla camera viene regolata per mezzo della rotella del mouse.

I parametri che consentono di configurare il comportamento di questo controller sono gli stessi di *IFRaySelectionMouseController*. In questo caso è però possibile specificare un ulteriore parametro, **wheelSensibility**, per regolare la sensibilità della rotella del mouse.

Kinect controller

L'interazione tramite Kinect, possibile grazie alla classe *IFCursorSelectionKinectController*, avviene in maniera del tutto analoga a quanto visto per la metafora *IFRaySelection*. Tuttavia, poiché in questo caso si utilizza un cursore anziché un raggio, c'è la possibilità che l'utente non riesca ad estendere il braccio destro a sufficienza per raggiungere l'oggetto da selezionare. Per tale ragione questo controller fornisce un meccanismo che simula un'estensione aggiuntiva del braccio, in analogia alla tecnica di interazione *"Go-Go"* (descritta nel paragrafo 1.6.1). L'utente può attivare l'estensione supplementare distendendo il braccio sinistro. L'entità dell'estensione è proporzionale alla distanza tra la posizione corrente della mano sinistra e la posizione in cui essa è stata precedentemente chiusa, causando l'attivazione del cursore.

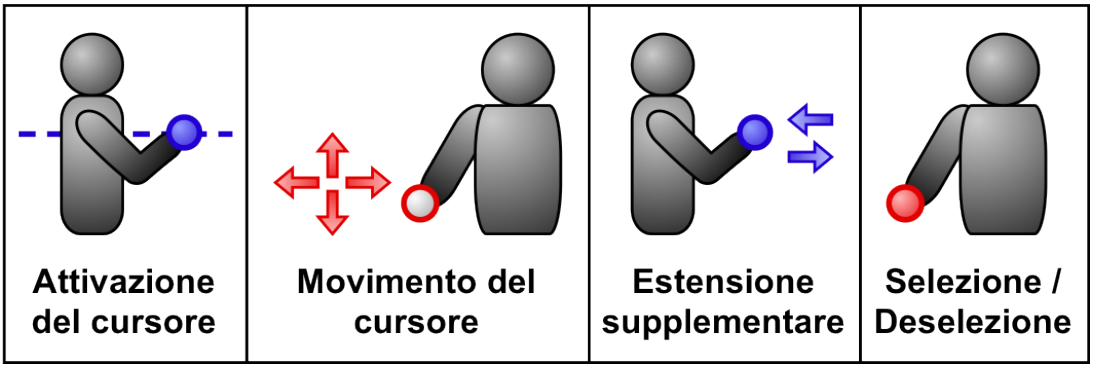


Figura 6.5: Uso del Microsoft Kinect per controllare la metafora *IFCursorSelection*

I principali parametri di questo controller sono:

- **testEveryFrame** - Ha la stesso scopo visto in precedenza per gli altri controller.
- **xyConversionFactor** e **zConversionFactor** - Sono parametri opzionali che consentono di regolare il fattore di conversione tra il sistema di riferimento locale all'utente (nel mondo reale) ed il sistema di coordinate della scena tridimensionale.
- **extensionScaleFactor** - Parametro opzionale che rappresenta il fattore di conversione usato per calcolare l'entità dell'estensione supplementare (e virtuale) del braccio destro.

Capitolo 7

Il package "GUI"

Questo package contiene le classi necessarie alla realizzazione di semplici interfacce grafiche bidimensionali. Esso comprende alcuni controlli grafici di base (come il *button*, il *toggle-button* ed il *panel*) ed un meccanismo di *layout* sufficientemente flessibile.

È bene precisare che questo modulo è da considerarsi un primo tentativo di integrare nel framework le funzionalità necessarie alla realizzazione di una semplice interfaccia grafica. Sebbene funzionante e utilizzabile già allo stato attuale, alcuni suoi aspetti possono essere migliorati in futuro.

7.1 Il pattern Model View Controller

Uno dei pattern architetturali più importanti nello sviluppo di applicazioni interattive è il *Model View Controller* (MVC). È stato formulato alla fine degli anni '70 da Trygve Reenskaug[1] e prevede che l'applicazione venga strutturata in tre parti distinte:

Model - Comprende i dati relativi al dominio dell'applicazione e la logica per la loro manipolazione.

Controller - Aggiorna il modello in risposta all'input dell'utente.

View - Fornisce una rappresentazione visiva dei dati contenuti nel modello.

Questo pattern viene usato non solo per la definizione dell'architettura dell'applicazione, ma anche a livello di singolo componente grafico. L'approccio tradizionale prevede che ogni controllo grafico venga implementato come un'entità autosufficiente, nella quale i tre ruoli previsti dal MVC si riflettono solo nell'organizzazione interna del codice. Con lo

[1] Pagina Web di Trygve Reenskaug presso l'Università di Oslo - http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html

sviluppo delle interfacce grafiche e con l'introduzione di *widget* sempre più complessi, la distinzione tra i tre ruoli è diventata sempre più marcata. In Swing, ad esempio, vi è una separazione esplicita tra il modello del controllo grafico e la coppia view-controller (chiamata *UI delegate*) che ne definisce il "*look & feel*"[2]. Questo consente sia di definire un modello personalizzato per il controllo, sia modificarne il "*look & feel*".

Approcci di questo tipo, però, mal si conciliano con le finalità di questo framework, per i seguenti motivi:

- Poiché i controlli grafici tipicamente usati nelle applicazioni tridimensionali interattive sono piuttosto semplici, la separazione del modello dal resto del componente grafico non è fondamentale (sebbene eventualmente possibile).
- La stretta correlazione tra *view* e *controller* è incompatibile con l'obiettivo del framework di rendere le varie funzionalità indipendenti dal tipo di dispositivi di input usati.

Per tali ragioni, nella progettazione di questo sotto-sistema si è scelto un approccio differente, che prevede la netta separazione dei controlli grafici dagli oggetti controller che ne modificano lo stato in risposta all'input di uno specifico dispositivo. La definizione del modello e della vista (costituita dal metodo *Draw*) restano invece competenza del *widget*.

7.2 La classe IFWidget

IFWidget è la superclasse di tutti i controlli grafici, siano essi componenti interattivi (ad esempio il bottone) o semplici contenitori (come i pannelli).

Ogni controllo ha un identificatore unico (di tipo intero o stringa) ed è caratterizzato dalle seguenti proprietà di base:

Enabled - Se il controllo è disabilitato il suo stato interno non può essere modificato e, di conseguenza, non emette eventi (se previsti).

Visible - Indica se il controllo deve essere disegnato o meno. Un controllo "invisibile" non risponde agli stimoli esterni, in quanto implicitamente disabilitato.

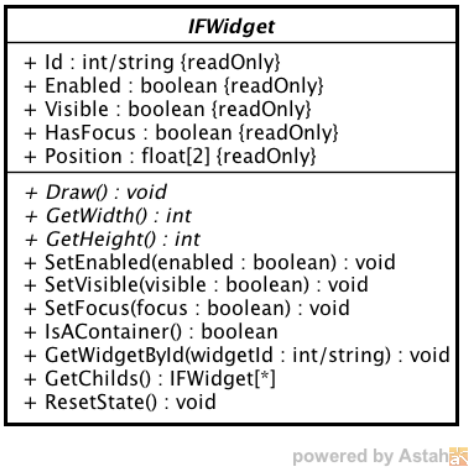


Figura 7.1: La classe *IFWidget*

[2] A Swing Architecture Overview, Oracle Technology Network - http://www.oracle.com/technetwork/java/architecture-142923.html

Focus - Indica se il componente ha il *focus*. Cosa questo comporti dipende dallo specifico controllo e soprattutto dal tipo dei dispositivi usati per interagire con la GUI. Ad esempio, usando un dispositivo di puntamento il focus serve generalmente solo a fornire un feedback visivo. Assume invece un ruolo decisamente più importante con dispositivi quali la tastiera od il joypad, in quanto consente di evidenziare il controllo al quale verrà recapitato l'input.

Per ognuna di queste proprietà *IFWidget* definisce un apposito campo (accessibile in sola lettura) ed un metodo per impostarne il valore.

Forse il metodo più importante dichiarato in questa classe è **Draw**, che ha il compito di disegnare a schermo il componente grafico (e gli altri controlli in esso contenuti, se presenti). Per ovvie ragioni *Draw* è un metodo astratto, quindi deve essere implementato da ogni sottoclasse concreta di *IFWidget*.

La posizione nella quale il componente verrà disegnato è contenuta nel campo **Position**[3]. Si noti che non è possibile posizionare direttamente il controllo. Questo deriva dalla scelta di imporre che ogni *widget* debba essere inserito all'interno di un contenitore[4]. Al momento dell'inserimento, la posizione viene specificata in relazione al contenitore e non in termini assoluti. È poi compito di quest'ultimo calcolare (ogni volta che viene invocato il suo metodo *Draw*) la posizione assoluta di ogni suo figlio, in funzione della politica di layout adottata e della dimensione corrente della finestra. Per quanto riguarda le dimensioni del controllo grafico, sono previsti i metodi **GetWidth** e **GetHeight**.

IFWidget dichiara anche un insieme di metodi dedicati ai "*container*":

- **IsAContainer** restituisce *true* se l'oggetto in questione può contenere al suo interno altri controlli grafici, *false* altrimenti.
- **GetWidgetById** consente di reperire un riferimento al componente figlio a partire dal suo Id. Se il controllo in questione non contiene figli, o se la ricerca ha un esito negativo, viene restituito *NULL*.
- **GetChilds** restituisce un array contenente i riferimenti ai figli.

Infine è presente il metodo **ResetState**, avente la stessa funzione del suo omonimo dichiarato da *IFController* (si veda il paragrafo 3.4), ovvero quello di riportare lo stato interno dell'oggetto alle sue condizioni iniziali.

Nella definizione di una classe concreta di *IFWidget* occorre:

[3] Poiché l'origine del sistema di riferimento bidimensionale si trova nell'angolo in alto a sinistra della finestra, con l'asse Y rivolto verso il basso, per posizione di un componente grafico si intende la posizione del vertice in alto a sinistra della regione rettangolare all'interno della quale viene disegnato il controllo grafico.

[4] Come vedremo in uno dei paragrafi successivi, al vertice di questa gerarchia si trova l'oggetto "*Window*" gestito dal GUI manager.

- Implementare i metodi astratti *Draw*, *GetWidth* e *GetHeight*.
- Richiedere come parametro del costruttore un identificatore e memorizarlo nel campo **Id**.
- Ridefinire se necessario il metodo *ResetState*.
- Se il controllo è un "*container*", devono essere ridefiniti anche i metodi *IsAContainer*, *GetChilds* e *GetWidgetById*. Si deve inoltre valutare l'opportunità di ridefinire *SetEnabled* e *SetVisible*, in modo da propagare la richiesta ai figli. Per gli altri tipi di componenti grafici tutto questo non è necessario.

7.3 Le classi IFPushButton e IFImagePushButton

Il *widget* più usato e conosciuto è il "bottone". La classe astratta *IFPushButton* definisce il modello di un generico bottone. Il suo stato interno, costituito dal campo **Pressed**, si limita a tener traccia del suo attuale stato di pressione. Il bottone può essere premuto e rilasciato rispettivamente chiamando i metodi **Press** e **Release**. Al termine di ogni pressione viene generato un evento **Push**. I listener di questo evento devono implementare il metodo

void OnPush(id)

il cui unico parametro consiste nell'identificatore del controllo grafico che lo ha emesso.

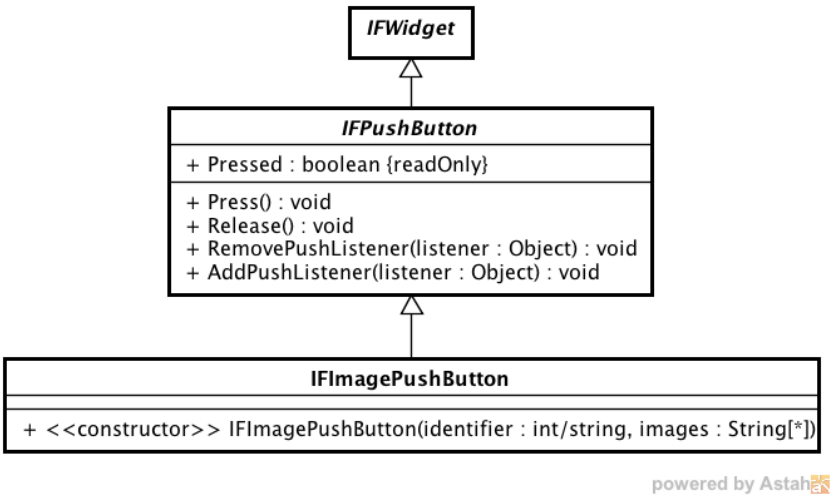


Figura 7.2: Le classi *IFPushButton* e *IFImagePushButton*

IFPushButton non specifica come deve essere disegnato il bottone, lasciando il compito di implementare il metodo *Draw* alle sue sottoclassi concrete. Nel framework è attualmente presente una sola sottoclasse, *IFImagePushButton*, che disegna il controllo a partire da un

insieme di immagini. Ogni immagine è associata ad un diverso stato del bottone (normale, premuto, focus, disabilitato ecc.).

7.4 Le classi IFToggleButton e IFImageToggleButton

A differenza del "*push button*", che resta premuto fintanto che l'utente agisce su di esso (comportandosi quindi come un pulsante), il "*toggle button*" ha due stati stabili e passa dall'uno all'altro in seguito ad una pressione (in analogia con l'interruttore, dal quale prende il nome). I due stati vengono qui chiamati "*checked*" e "*unchecked*".

La classe astratta *IFToggleButton* definisce il modello di un bottone di questo tipo. Il suo stato corrente può essere verificato tramite il campo **IsChecked**. Come per *IFPush-Button*, i metodi **Press** e **Release** consentono rispettivamente di premere e rilasciare il bottone, provocando il cambiamento di stato. In alternativa si può passare direttamente da uno stato all'altro chiamando il metodo **Toggle**, anche se in genere è preferibile usare *Press/Release* in modo da poter dare un feedback visivo dell'operazione.

Quando il bottone passa dallo stato "*unchecked*" a "*checked*" viene generato l'evento **Checked**, mentre a seguito della transizione inversa viene emesso l'evento **Unchecked**. I metodi per la loro gestione sono rispettivamente

void OnChecked(id)

e

void OnUnchecked(id)

dove il parametro "*id*" è l'identificatore del controllo grafico che lo ha emesso.

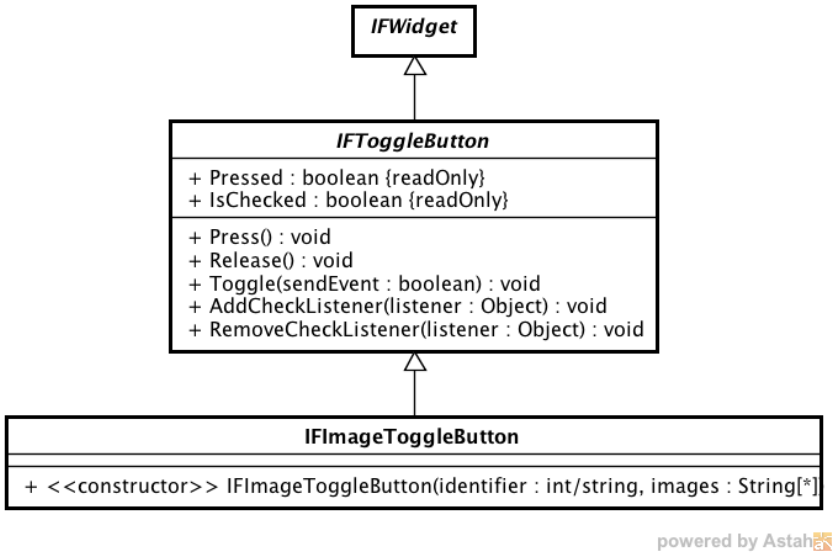


Figura 7.3: Le classi *IFToggleButton* e *IFImageToggleButton*

La sottoclasse concreta *IFImageToggleButton* contiene il codice necessario a disegnare il bottone a partire da un insieme di immagini, che ne definiscono l'aspetto nei suoi vari stati.

7.5 La classe IFImageBox

La classe *IFImageBox* rappresenta una regione rettangolare dello schermo all'interno della quale viene disegnata un'immagine. Non si tratta di un controllo interattivo, quindi il suo utilizzo ha tipicamente finalità estetiche. Può anche risultare utile per realizzare delle semplici "*label*", in alternativa alla funzione *ConsoleText* di XVR (che presenta diverse limitazioni).

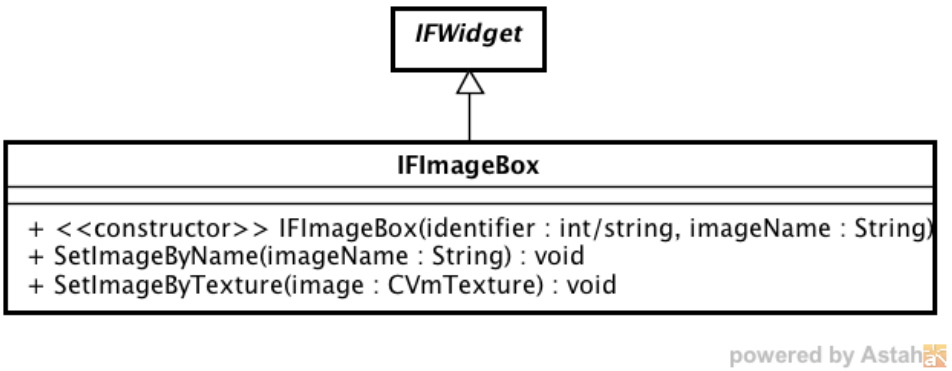


Figura 7.4: La classe *IFImageBox*

L'immagine da disegnare deve essere specificata come parametro del costruttore. È comunque possibile cambiarla in un secondo momento attraverso i metodi **SetImageByName** e **SetImageByTexture**.

7.6 Le classi IFPanel, IFFilledPanel e IFTexturedPanel

IFPanel implementa un generico contenitore di controlli grafici. Il "pannello" può avere una dimensione prefissata oppure può ridimensionarsi automaticamente in accordo alla finestra che lo contiene. Questa classe implementa anche una semplice politica di *layout* dei controlli.

Le principali caratteristiche del pannello vengono stabilite in fase di creazione. I parametri del costruttore sono infatti:

identifier - L'identificatore associato al controllo. Può essere di tipo intero o stringa.

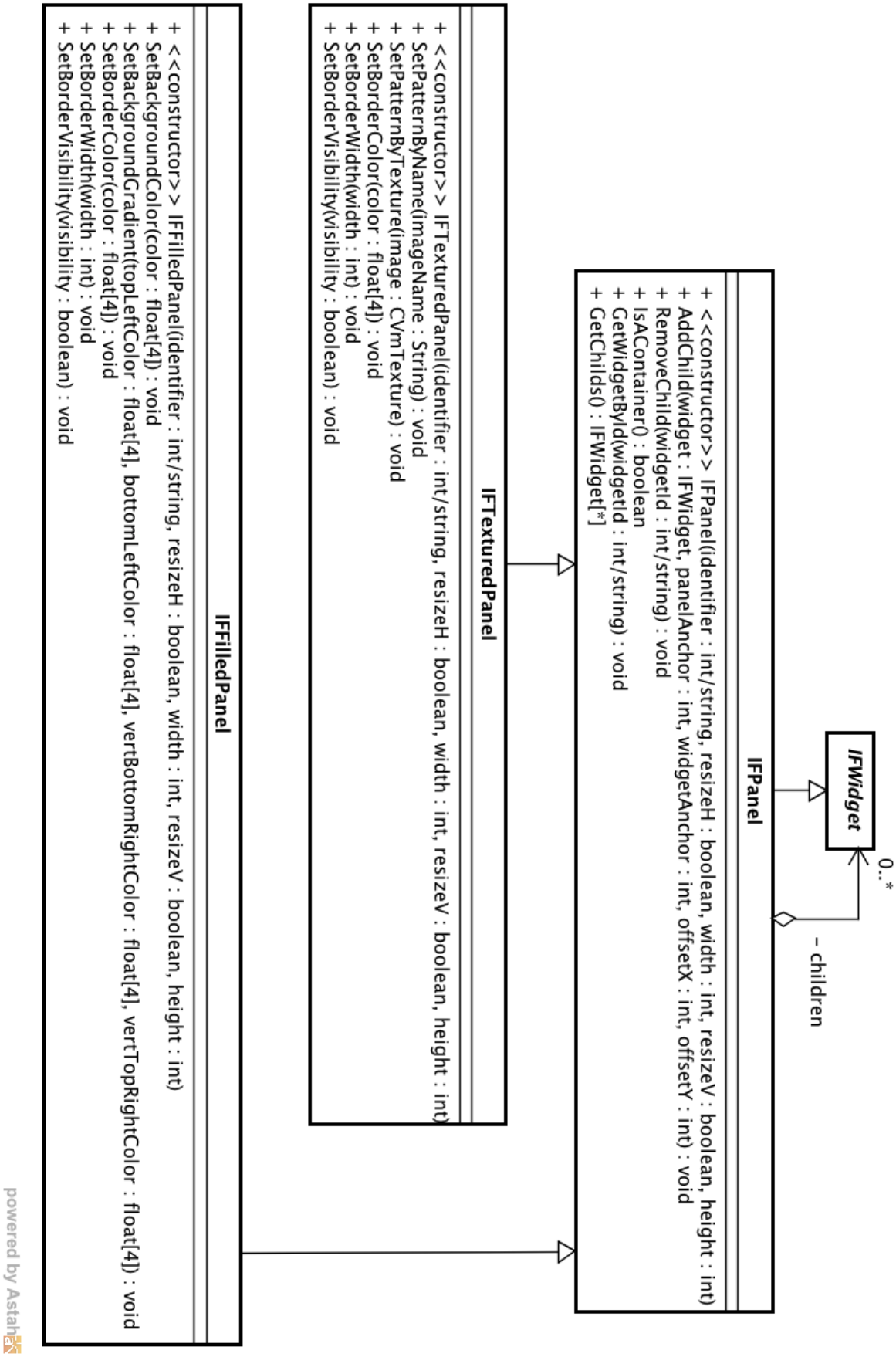


Figura 7.5: Le classi *IFPanel*, *IFFilledPanel* e *IFTexturedPanel*

resizeHorizontally - Flag booleano usato per stabilire se la larghezza del pannello deve restare costante o se può adattarsi dinamicamente alle dimensioni della finestra principale.

width - Rappresenta la larghezza del pannello. Se il pannello non può ridimensionarsi orizzontalmente questo parametro esprime la sua larghezza in pixel. In caso contrario si deve specificare un valore in virgola mobile compreso nell'intervallo (0,1], che esprime la larghezza come frazione di quella della finestra.

resizeVertically- Flag booleano usato per stabilire se l'altezza del pannello deve restare costante o se può adattarsi dinamicamente alle dimensioni della finestra principale.

height - Rappresenta l'altezza del pannello, espressa in pixel o come frazione dell'altezza della finestra principale.

La politica di layout implementata da IFPanel è nota come "*Anchor Layout*". Il nome deriva dal fatto che ad ogni controllo grafico sono associati nove "*punti di ancoraggio*" (detti anche "*ancore*"). Tali punti sono situati ai vertici del componente, a metà di ogni suo lato ed al centro dello stesso. Quando un componente viene inserito nel pannello, si scelgono due ancore (una per il pannello ed una per il controllo) e si specifica la distanza che deve essere mantenuta tra le due. Così facendo, quando la finestra viene ridimensionata, la posizione relativa tra i due resta inalterata.

L'inserimento avviene attraverso il metodo **AddChild** che, in accordo a quanto appena spiegato, richiede i seguenti parametri:

widget - Il controllo grafico da inserire. Può essere a sua volta un pannello.

panelAnchorPoint - Il punto di ancoraggio scelto per il contenitore.

widgetAnchorPoint - Il punto di ancoraggio scelto per l'oggetto da inserire.

offsetX - La distanza tra le due ancore lungo la componente X.

offsetY- La distanza tra le due ancore lungo la componente Y.

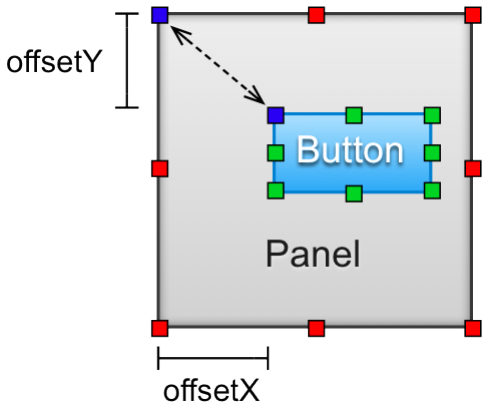


Figura 7.6: Layout mediante ancore

Il metodo **RemoveChild** esegue l'operazione opposta, rimuovendo dal contenitore il controllo avente l'identificatore passato come parametro.

IFPanel ridefinisce i metodi **SetEnabled**, **SetVisible** e **ResetState** (dichiarati da *IFWidget*), in modo da inoltrare la richiesta ai figli (dopo aver aggiornato il suo stato interno). Lo stesso avviene per **Draw**, dove però la fase di disegno è preceduta dal calcolo della posizione dei componenti figli, tenendo conto della dimensione corrente della finestra principale e dei parametri di layout associati a ciascun controllo. Il pannello in sé non viene disegnato e risulta quindi invisibile.

Il framework comprende altri due tipi di pannelli, definiti dalle classi *IFFilledPanel* e *IFTexturedPanel*, che estendono *IFPanel* sovrascrivendo il metodo *Draw*. Entrambe le classi implementano un pannello caratterizzato da sfondo e bordi visibili. Nel caso di *IFFilledPanel* lo sfondo è costituito da un gradiente o da un colore a tinta unita, assegnabili rispettivamente tramite i metodi **SetBackgroundColor** e **SetBackgroundGradient**. La classe *IFTexturedPanel* disegna invece un'immagine di sfondo, impostata dai metodi **SetPatternByName** e **SetPatternByTexture**. Se la dimensione del pannello supera quella dell'immagine, quest'ultima viene ripetuta fino a coprire l'intera area del pannello. Entrambe le classi consentono infine di disegnare un bordo, le cui caratteristiche vengono regolate dai metodi **SetBorderColor** e **SetBorderWidth**.

7.7 Le classi "controller"

Al fine di disaccoppiare l'interfaccia grafica dagli specifici dispositivi usati per interagire con essa, la gestione degli eventi di input non viene svolta dai *widget*, bensì da oggetti controller ad essi associati.

I controller dell'interfaccia grafica devono estendere la classe astratta *IFWigetController* (anziché *IFController*), mostrata in figura 7.7. Uno dei concetti chiave introdotti da questa classe è lo stato "*engaged*"[5] (occupato, in uso). Un controller si trova in stato "*engaged*" se è attualmente impegnato a gestire una sequenza di eventi di input. Fintanto che si trova in questo stato, tutti gli eventi generati dai dispositivi dovranno essere notificati soltanto a lui. In particolare, tali eventi non dovranno essere propagati ai controller che agiscono sull'ambiente virtuale. Segue che in ogni istante dovrà essere presente al più un controller in stato "*engaged*". Il metodo **IsEngaged** consente di conoscere lo stato attuale del controller.

Per comprendere meglio la necessità di un simile meccanismo, consideriamo una semplice applicazione dotata di interfaccia grafica e nella quale l'utente naviga all'interno della scena utilizzando il mouse. Supponiamo inoltre che l'utente prema il pulsante sinistro del

[5]Il nome è stato scelto in conformità con *EasyGUI*, una semplice libreria per realizzare interfacce grafiche in XVR. Il motivo per cui si è scelto di definire una nuova libreria di questo tipo, anziché usare *EasyGUI*, deriva dal fatto che quest'ultima supporta solo il mouse come dispositivo e input, e mal si integra con le altre meccaniche del framework.

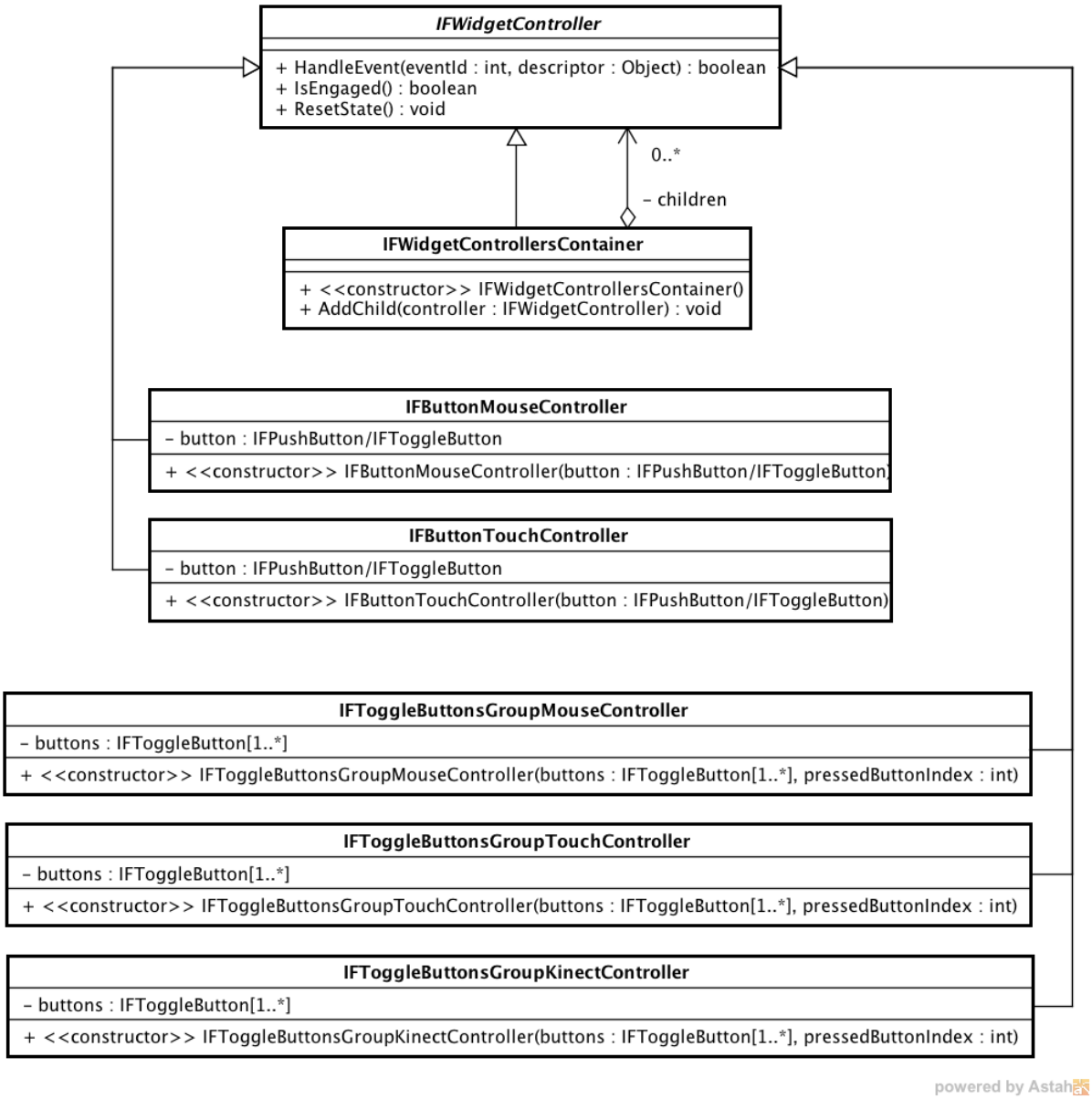


Figura 7.7: Le classi "*controller*"

mouse sopra ad un componente grafico. Da quel momento fino al rilascio del pulsante, tutti gli eventi di input dovranno confluire al solo controller del *widget* interessato. Infatti, se venisse notificato anche il controller che gestisce la metafora di navigazione, ciò produrrebbe una modifica della posizione e/o dell'orientamento della camera, mentre l'intenzione dell'utente era quella di agire soltanto sulla GUI. Per le stesse ragioni gli eventi non devono essere inoltrati al resto dell'interfaccia grafica.

Il metodo più importante dichiarato da *IFWigetController* è **HandleEvent** che ha il compito di gestire un evento di input generato da un dispositivo. I parametri del metodo sono rispettivamente l'identificatore intero dell'evento e un descrittore di tipo qualsiasi, compreso *NULL*. Tipicamente, l'implementazione di questo metodo (fornita dalle sottoclassi concrete di *IFWigetController*) consiste nei seguenti passi:

- Verificare se l'evento deve essere scartato (perché relativo ad un altro dispositivo oppure perché, semplicemente, la sua gestione non è prevista da questo controller).
- Gestire l'evento modificando lo stato del *widget* associato.
- Restituire *"this"* se il controller è entrato in stato "*engaged*" o se persiste questo stato, restituire *NULL* in caso contrario. Si noti comunque che quando un controller notifica al GUI manager[6] di essere uscito dallo stato "*engaged*" (restituendo *NULL*), il manager continuerà a considerarlo tale fino al frame successivo.

Infine, il metodo **ResetState** si occupa di riportare lo stato interno del controller e del widget da lui gestito nelle loro condizioni iniziali.

Attualmente nel framework sono definiti i seguenti controller per *widget*:

- **IFButtonMouseController** e **IFButtonTouchController**: aggiornano lo stato di un singolo "*push button*" o "*toggle button*" gestendo gli eventi di mouse e touchscreen.
- **IFToggleButtonsGroupMouseController**, **IFToggleButtonsGroupTouchController** e **IFToggleButtonsGroupKinectController**: consentono di costituire un "*toggle group*", cioè un insieme di "*toggle button*" nel quale solo uno dei bottoni che ne fanno parte può trovarsi in stato "*checked*". Gestiscono rispettivamente gli eventi di mouse, touchscreen e Kinect. Utilizzando il Kinect, l'utente può selezionare il bottone successivo eseguendo una gesture di tipo "*Swipe down*".
- **IFWidgetControllersContainer**: rappresenta una semplice lista di *IFWigetController*. Implementa anch'esso l'interfaccia dichiarata da *IFWigetController*, in modo da poter essere trattato nello stesso modo di un singolo controller[7].

[6] Di cui parleremo al paragrafo 7.8.
[7] In altri termini, è stato applicato il pattern *Composite* [16].

7.8 La classe IFGUIManager

Il GUI manager ha lo scopo di gestire i componenti che implementano le funzionalità dell'interfaccia grafica e di fornire un'interfaccia centralizzata attraverso la quale è possibile integrare questo sottosistema al il resto del framework.

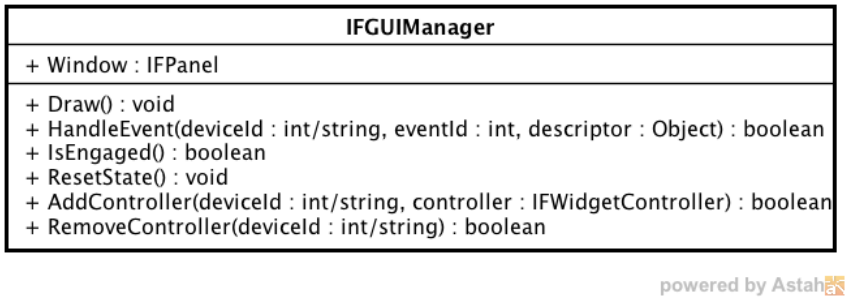


Figura 7.8: La classe *IFGUIManager*

In particolare:

- Mette a disposizione attraverso il campo **Window** il "pannello radice", all'interno del quale devono essere inseriti tutti i componenti grafici. È un'istanza di *IFPanel* e copre l'intera area della finestra del player XVR. Il suo identificatore deve essere definito all'inizio del programma per mezzo della costante IFC_GUI_MANAGER_WINDOW_ID, in maniera analoga a quanto fatto per stati e modalità di interazione (si veda il listato 2.1 a pagina 39).

- Fornisce il metodo **Draw** per disegnare la GUI. L'implementazione di questo metodo si occupa settare opportunamente lo stato di OpenGL e di invocare l'omonimo metodo di *Window*.

- Gestisce un dizionario nel quale vengono associati i dispositivi ai rispettivi controller. Il metodo **AddController** consente di registrare un oggetto controller presso il GUI manager, specificando l'identificatore del dispositivo dal quale desidera ricevere gli eventi. È possibile associare un singolo controller per ogni dispositivo, per cui in genere si ricorre ad un oggetto di tipo *IFWidgetControllersContainer* per raggruppare tutti i controller necessari. Il metodo **RemoveController** si occupa invece di rimuovere una associazione dispositivo-controller dal dizionario.

- Fornisce il metodo **HandleEvent** attraverso cui un dispositivo può notificare un evento all'interfaccia grafica. Questo metodo restituisce *true* se esiste un controller attualmente in stato "*engaged*", *false* in caso contrario. Il dispositivo deve controllare il valore restituito ed inoltrare l'evento ai suoi listener solo se il risultato è *false*. L'implementazione di questo metodo si occupa di reperire dal dizionario il controller

associato al dispositivo che ha generato l'evento, e di notificargli l'evento. Se però esiste un controller attualmente in stato "*engaged*" (cosa di cui il GUI manager tiene traccia), l'evento sarà inviato solo a quest'ultimo.

- Fornisce il metodo **ResetState** per "resettare" l'interfaccia grafica al suo stato iniziale.

È importante notare che solo un sottoinsieme di questi metodi è destinato al programmatore. Infatti, i metodi **Draw** e **ResetState** vengono invocati automaticamente dall'oggetto "applicazione", rispettivamente ad ogni frame ed in presenza di un cambiamento di stato. L'uso del metodo **HandleEvent** è invece destinato ai soli oggetti "dispositivo".

Tipicamente al programmatore è richiesto solo di creare i *widged* ed i relativi *controller*, e di "inserirli" nel GUI manager attraverso il pannello principale (*Window*) ed il metodo *AddController*. In alcuni casi particolari può anche invocare il metodo *ResetState*, se lo ritiene necessario. È certamente compito del programmatore istanziare il GUI manager e registrarlo presso l'oggetto "applicazione" attraverso il metodo **SetGUIManager** della classe IFApplication. Può esistere al più un GUI manager per ogni stato dell'applicazione, ed è possibile ottenere un riferimento a quello corrente grazie al metodo **GetGUIManager** della classe IFApplication.

7.9 Supporto della GUI da parte dei dispositivi

Se si ha la necessità di definire una classe per supportare un nuovo dispositivo, si può prendere in considerazione di renderla compatibile con i meccanismi fin qui descritti. Per far ciò basta rispettare qualche semplice regola:

1. Definire un identificatore intero per ogni evento generato dal dispositivo. Ad esempio, gli identificatori definiti dalla classe *IFKeyboard* sono i seguenti:

```
#define IFC_KEYBOARD_EVT_KEY_DOWN    201
#define IFC_KEYBOARD_EVT_KEY_UP   202
```

2. Prima di notificare un evento ai listener, è necessario inviarlo al GUI manager e verificare se l'evento è stato "catturato" o meno dall'interfaccia grafica. Se ciò accade, l'evento non deve essere propagato ulteriormente. Il seguente frammento di codice fornisce un esempio su come procedere:

```
var captured = false;
var gui = IFAPP.GetGUIManager();
```

```
if(gui != NULL)
   captured = gui.HandleEvent(this.Id, eventId, descriptor);

if(!captured) {
   for(var i = 0; i < len(_listeners); i++)
      _listeners[i].OnMyEvent(descriptor);
}
```

7.10 Un semplice esempio

Concludiamo il capitolo con un semplice esempio nel quale vengono mostrati i passi necessari per la creazione dell'interfaccia grafica. Il codice di esempio crea un pannello e due bottoni, posizionandoli rispettivamente nella parte superiore ed inferiore del pannello. Viene inoltre definito un semplice listener che stampa un messaggio a schermo a seguito della pressione dei bottoni.

```
// Semplice listener che stampa un messaggio
// su console quando viene premuto un bottone
class SimpleButtonListener {
   OnPush(widgetId);
   OnChecked(id);
   OnUnchecked(widgetId);
}

function SimpleButtonListener::OnPush(widgetId){
   Outputln("Button␣", widgetId, "␣pushed");
}

function SimpleButtonListener::OnChecked(widgetId){
   Outputln("Button␣", widgetId, "␣checked");
}

function SimpleButtonListener::OnUnchecked(widgetId){
   Outputln("Button␣", widgetId, "␣unchecked");
}

[...]

function buildTheGUI(mouse) {
   // Crea un push button ed un toggle button.
   var button1 = IFImagePushButton("button1", "button1Normal.png", "
      button1Disabled.png", "button1Pressed.png", "button1Hover.png");
```

```
var button2 = IFImageToggleButton("button2", "button2Checked.png", "
    button2CheckedHover.png", "button2Unchecked.png", "
    button2UncheckedHover.png", "button2Disabled.png");

// Crea un pannello di larghezza pari a 100 pixel
// e di altezza pari a quella della finestra.
var leftPanel = IFPanel("leftPanel", false, 100, true, 1.0);

// Si aggiungono i due bottoni al pannello.
// Per posizionare il primo, si specifica che il suo vertice
// superiore sinistro deve restare "ancorato" al vertice
// superiore sinistro del pannello, ad una distanza di
// 10px lungo l'asse X e 10px lungo l'asse Y.
// Il secondo bottone viene vincolato al vertice inferiore
// sinistro, alla stessa distanza.
leftPanel.AddChild(button1, IFC_ANCHOR_TOP_LEFT, IFC_ANCHOR_TOP_LEFT,
    10, 10);
leftPanel.AddChild(button2, IFC_ANCHOR_BOTTOM_LEFT,
    IFC_ANCHOR_BOTTOM_LEFT, 10, -10);

// Si creano i controller necessari ad interagire
// con i bottoni attraverso il mouse.
var button1Controller = IFButtonMouseController(button1);
var button2Controller = IFButtonMouseController(button2);

// I controller vengono inseriti in un contenitore,
// in modo da poter essere registrati presso il GUI manager.
var controllers = IFWidgetControllersContainer();
controllers.AddChild(button1Controller);
controllers.AddChild(button2Controller);

// Crea il GUI manager.
var gui = IFGUIManager();

// Aggiunge il pannello prima creato al pannello
// principale, vincolato al lato sinistro della finestra.
gui.Window.AddChild(leftPanel, IFC_ANCHOR_TOP_LEFT, IFC_ANCHOR_TOP_LEFT,
     0, 0);

// Registra la collezione di controller presso il
// GUI manager, specificando che siamo interessati a gestire
// i soli eventi del mouse.
gui.AddController(mouse.Id, controllers);

// Registra il GUI manager presso l'oggetto "applicazione".
IFAPP.SetGUIManager(guiMouse);
```

```
    // Si istanzia il semplice listener definito in precedenza
    // e lo si registra presso i bottoni.
    var listener = SimpleButtonListener();
    button1.AddPushListener(listener);
    button2.AddCheckListener(listener);
}
```

Capitolo 8

Il package “Intersection”

In questo package cono contenute le strutture dati e gli algoritmi usati per eseguire alcuni tipi di test di intersezione tra primitive geometriche. Questi algoritmi sono alla base dell'implementazione di funzionalità quali il *terrain following* e la *collision detection* della camera, oltre che delle metafore di selezione.

In linea teorica questi test potrebbero essere eseguiti direttamente sulla stessa geometria usata per il rendering della scena. Tuttavia si deve considerare che i modelli di rendering sono generalmente costituiti da un elevato numero di poligoni e, dovendo esaminare ognuno di essi (il procedimento ha complessità lineare nel numero di poligoni), il costo complessivo del procedura risulterebbe troppo gravoso per le applicazioni in tempo reale. Una prima soluzione al problema consiste nel creare una scena “gemella”, ottenuta sostituendo la geometria originale con primitive geometriche più elementari (ad esempio: sfere, parallelepipedi ecc.). Queste primitive vengono dette “*bounding volume*”, in quanto usate per incapsulare uno o più elementi geometrici di natura più complessa [11].

Alcuni dei *bounding volume* più comuni sono la *sfera*, l'*axis-aligned bounding box*[1] (AABB), l'*oriented bounding box*[2] (OBB), la *capsula*[3] ed il *discrete-orientation polytopes* (DOP)[4]. Ognuno di essi ha caratteristiche differenti in termini di costo degli algoritmi di intersezione e di “aderenza”[5] all'oggetto in esso contenuto.

I *bounding volume* supportati dal framework e le relative classi sono:

Sfera - Implementata dalla classe *IFSphere*, descritta al paragrafo 8.4.

Capsula - Implementata dalla classe *IFCapsule*, descritta al paragrafo 8.5.

[1] Parallelepipedo rettangolo orientato con gli assi del sistema di riferimento.
[2] Parallelepipedo rettangolo con orientamento arbitrario.
[3] Cilindro con due semisfere alle estremità.
[4] Per maggiori dettagli su questi (ed altri) *bounding volume* e sulle loro caratteristiche si può consultare [11].
[5] Per “aderenza” si intende la differenza di volume tra un oggetto ed il *bounding volume* che lo racchiude. Dipende sia dal tipo di *bounding volume* che dalla forma dell'oggetto.

AABB - Implementato dalla classe *IFAABB*, descritta al paragrafo 8.6.

OBB - Implementato dalla classe *IFOBB*, descritta al paragrafo 8.7.

Una volta racchiusi gli elementi della scena all'interno di *bounding volume* si procede come segue:

- I test vengono eseguiti (almeno inizialmente) sui *bounding volume*, per i quali possono essere sfruttati algoritmi molto efficienti.
- Se il test fallisce per un certo *bounding volume*, esso fallirà certamente anche per l'elemento in esso racchiuso, che quindi non dovrà essere esaminato. Segue che tanto più il *bounding volume* "aderisce" ad un elemento, maggiore sarà la possibilità che il test dia un esito negativo.
- Se invece il test ha esito positivo si può scegliere, in base alle necessità, di accontentarsi delle informazioni raccolte con il *bounding volume* oppure di eseguire il test anche sulla geometria in esso contenuta. Poiché il numero di *bounding volume* per i quali un dato test dà esito positivo è in genere molto minore del loro numero complessivo, il numero di poligoni da esaminare sarà comunque una piccola frazione di quello complessivo.

Sebbene questa tecnica introduca un sensibile miglioramento delle prestazioni, la sua complessità asintotica resta lineare.

Si può allora pensare di ricorrere ad una struttura ad albero, detta *Bounding Volume Hierarchy* (BVH), nella quale i nodi foglia sono i *bounding volume* che racchiudono gli elementi della scena. Anche i nodi interni sono costituiti da *bounding volume*, aventi dimensione tale da incapsulare tutte le foglie del loro sotto-albero. Il test di intersezione viene quindi eseguito sull'albero, con il vantaggio che, se il test di intersezione per un nodo interno fallisce, il suo intero sotto-albero non verrà esaminato. Si ottiene così una complessità sublineare nel caso medio.

La classe *IFAABBTree*, descritta al paragrafo 8.9, implementa un BVH binario nel quale ad ogni nodo interni è associato un axis-aligned bounding box.

Attualmente nel framework sono definiti gli algoritmi relativi a tre tipi di test:

1. Intersezione con un raggio
2. Intersezione con una sfera statica
3. Intersezione con una sfera in movimento

Gli algoritmi scelti sono tra i più noti ed usati in questo campo e vengono descritti nel dettaglio in [11, 10, 27][6]. Di essi viene fornita una implementazione in linguaggio C ed

[6] Nella documentazione del framework viene riportato un riferimento più specifico per ognuno di loro.

una in linguaggio S3D. La prima offre le prestazioni migliori ed è rivolta alle applicazioni per sistemi desktop o immersivi, mentre la seconda può essere impiegata anche nelle applicazioni Web-based.

8.1 La classe IFRay

La classe *IFRay* modella la primitiva geometrica raggio ("*ray*"). Formalmente un raggio è una semiretta, tuttavia nella pratica è utile ricorrere a raggi di lunghezza finita, ovvero a **segmenti orientati** [27, 9].

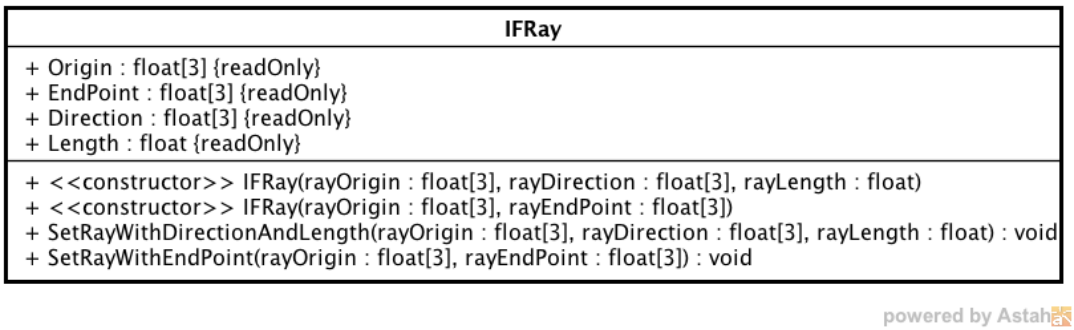


Figura 8.1: La classe IFRay

Il raggio può essere definito per mezzo dei suoi estremi, oppure specificando un punto di origine, la lunghezza e la direzione. La classe *IFRay* espone un campo per ognuna di queste proprietà, accessibile in sola lettura. È possibile modificare il raggio in un secondo momento grazie ai metodi **SetRayWithDirectionAndLength** e **SetRayWithEndPoint**.

8.2 La classe IFIntersectableObject

La classe astratta *IFIntersectableObject* modella un generico elemento geometrico su cui possono essere eseguiti test di intersezione con altre primitive geometriche.

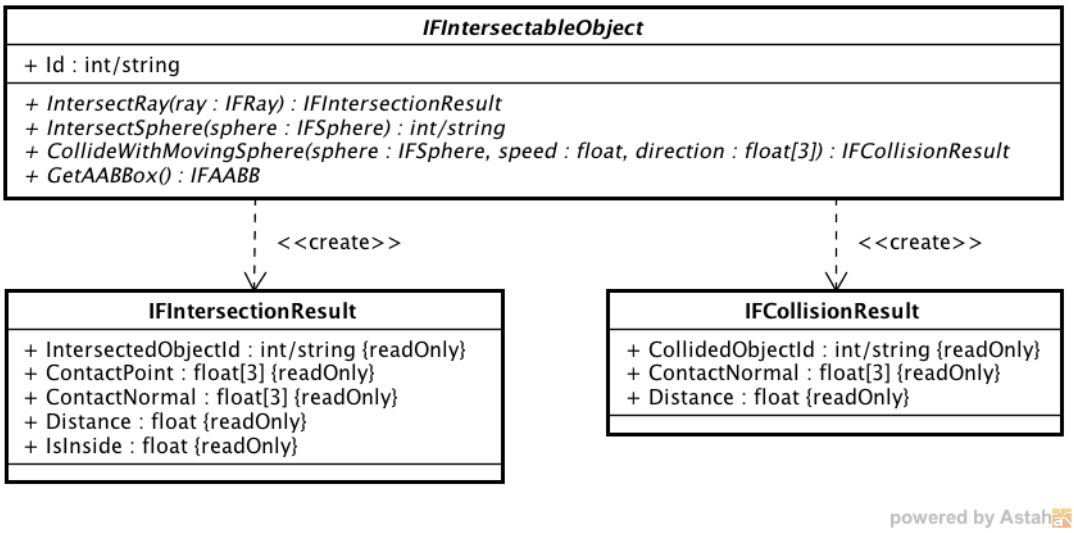


Figura 8.2: La classe IFIntersectableObject

Ad ogni tipo di test corrisponde un metodo (astratto) di *IFIntersectableObject*:

1. **Intersezione con un raggio**
   Viene eseguita dal metodo **IntersectRay**. Se l'algoritmo rileva un'intersezione tra l'oggetto ed il raggio (istanza di *IFRay*) passato come parametro, viene restituito un oggetto di tipo *IFIntersectionResult*, contenente:

   - L'identificatore dell'oggetto intersecato.
   - Il punto di intersezione.
   - La normale alla superficie nel punto di intersezione.
   - Un flag booleano che indica se il punto di origine del raggio si trova all'interno dell'oggetto.
   - La distanza tra l'origine ed il punto di intersezione. In generale è compresa nell'intervallo [0, lunghezza del raggio], salvo i casi in cui l'origine del raggio si trova all'interno dell'oggetto. In queste situazioni l'algoritmo determina i due punti (appartenenti alla superficie dell'oggetto) nei quali avviene l'intersezione con la retta su cui giace il raggio, e restituisce quello più vicino all'origine del raggio stesso. È quindi possibile che la distanza fornita come risultato sia negativa, o superiore alla lunghezza del raggio. Il modo in cui viene gestito questo caso particolare risulta utile per l'implementazione delle funzionalità di rilevamento e gestione delle collisioni.

   Se il test fallisce viene restituito *NULL*.

2. **Intersezione con una sfera statica**
   Corrisponde al metodo **IntersectSphere**. Gli algoritmi appartenenti a questa famiglia verificano la distanza tra il centro della sfera e la superficie dell'oggetto. Se tale distanza è inferiore al raggio della sfera i due oggetti si intersecano. Restituisce l'identificatore dell'oggetto intersecato oppure *NULL* (se il test fallisce).

3. **Intersezione con una sfera in movimento**
   Il metodo **CollideWithMovingSphere** verifica l'intersezione tra l'oggetto stesso (considerato statico) ed una sfera in movimento con moto rettilineo uniforme. La posizione della sfera durante lo spostamento viene espressa dalla seguente equazione :

$$\begin{cases} C(t) = C_0 + t \cdot \vec{v} \\ \quad t \,\epsilon\, [0,1] \end{cases}$$

   dove t rappresenta il parametro tempo (compreso tra 0 e 1), C_0 il centro della sfera prima dello spostamento, $\vec{v}$ il vettore velocità e $C(t)$ la posizione della sfera all'istante t. L'algoritmo deve determinare il tempo t_c nel quale le due primitive geometriche entrano in contatto, ovvero quando la distanza tra centro della sfera e la superficie dell'oggetto è pari al raggio della sfera stessa (come mostrato in figura 8.3). Viene restituita un'istanza di *IFCollisionResult*, contenente:

   - L'identificatore dell'oggetto con cui la sfera entra in contatto.
   - La normale alla superficie nel punto di contatto.
   - Un flag booleano che indica se il punto di origine del raggio si trova all'interno dell'oggetto.
   - La distanza che deve percorrere la sfera per entrare in contatto con l'oggetto. Rappresenta il termine $t \cdot \vec{v}$ dell'equazione precedente. Se sfera ed oggetto non sono in contatto tra loro all'inizio dello spostamento, la distanza sarà compresa nell'intervallo $[0, ||\vec{v}||]$. In caso contrario l'algoritmo deve determinare il più vicino punto di contatto ottenuto sposando la sfera lungo la retta cui appartiene la direzione di spostamento, e calcolare l'entità dello spostamento di conseguenza. Quando ciò accade, è possibile ottenere valori al di fuori dell'intervallo $[0, ||\vec{v}||]$.

   Se il test fallisce viene restituito *NULL*.

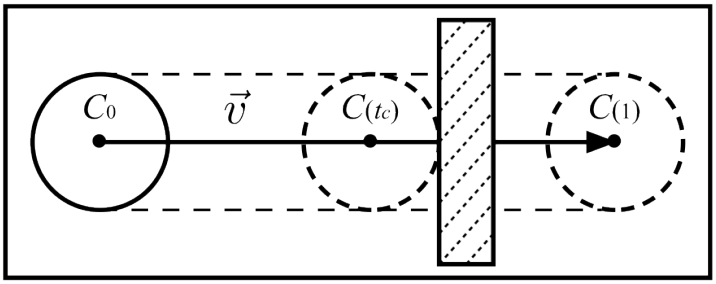


Figura 8.3: Test di intersezione tra un oggetto statico ed una sfera in movimento

È normale domandarsi perché i risultati dei metodi appena descritti comprendono anche l'identificatore dell'oggetto. La ragione risiede nel fatto che le sottoclassi di *IFIntersectableObject* non rappresentano soltanto primitive geometriche, ma anche collezioni (come nel caso della classe *IFAABBTree*). In altri termini, è stato applicato il pattern *Composite* [16].

Infine, la classe *IFIntersectableObject* dichiara il metodo **GetAABBox**, avente il compito di calcolare e restituire l'axis-aligned bounding box dell'oggetto in questione. Questo metodo è di fondamentale importanza nella costruzione di una gerarchia di bounding box (*IFAABBTree*), ma può risultare utile anche in altri contesti.

8.3 Le interfacce IFMovableObject e IFRotatableObject

L'interfaccia *IFMovableObject* dichiara i metodi che consentono di spostare un elemento geometrico nello spazio o di reperirne la posizione corrente. Viene implementata da tutti i *bounding volume* definiti in questo package, così come dai loro contenitori.

Analogamente, l'interfaccia *IFRotatableObject* dichiara i metodi che consentono di regolare l'orientamento di un oggetto o di ottenere informazioni su quello corrente. Nei metodi **SetRotation** e **Rotate**, le rotazioni devono essere espresse come una tripla di angoli di Eulero. **GetRotationMatrix** restituisce invece l'orientamento corrente rappresentato come una matrice di rotazione 16x16 [7]. Non tutti

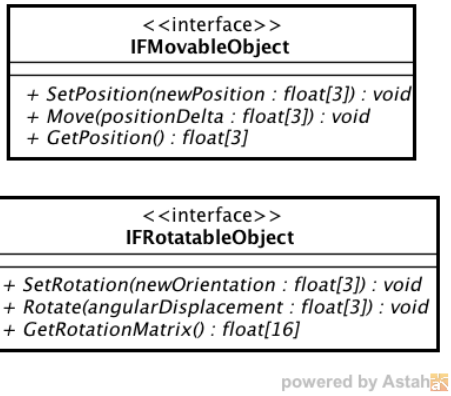


Figura 8.4: Le interfacce *IFMovableObject* e *IFRotatableObject*

[7] Nel formato usato da OpenGL.

i bounding volume implementano questa interfaccia. Ad esempio non avrebbe senso ruotare una sfera. Di contro sarebbe possibile ruotare un AABB, ma soltanto per multipli di 90^{o}.

8.4 La classe IFSphere

La classe *IFSphere* implementa un *bounding volume* sferico. La sfera è caratterizzata dalla posizione del centro e dal suo raggio, memorizzati rispettivamente nei campi **Center** e **Radius**. È possibile modificare direttamente questi campi.

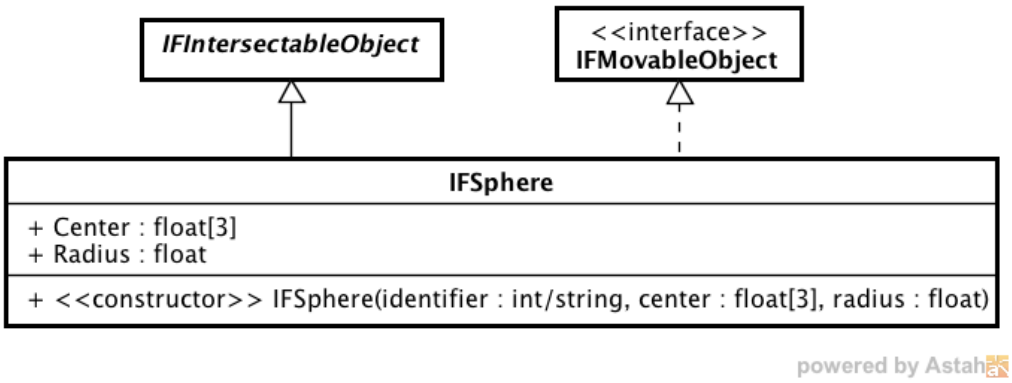


Figura 8.5: La classe *IFSphere*

8.5 La classe IFCapsule

Una capsula è costituita da un cilindro avente due semisfere alle estremità. La capsula è definita da un segmento $\overline{P_1P_2}$ (che determina l'asse longitudinale del cilindro e la sua altezza) e dal valore r del suo raggio. Segue che un punto X appartiene alla capsula se e solo se la sua distanza dal segmento $\overline{P_1P_2}$ è minore o uguale a r. Per ragioni pratiche alla capsula viene assegnata anche una direzione, ottenuta normalizzando il vettore $\overrightarrow{(P_2 - P_1)}$.

Una volta creata, è possibile modificare le caratteristiche della capsula attraverso i metodi **SetCapsuleByEndPoints** e **SetCapsuleByCenterAndAxis**.

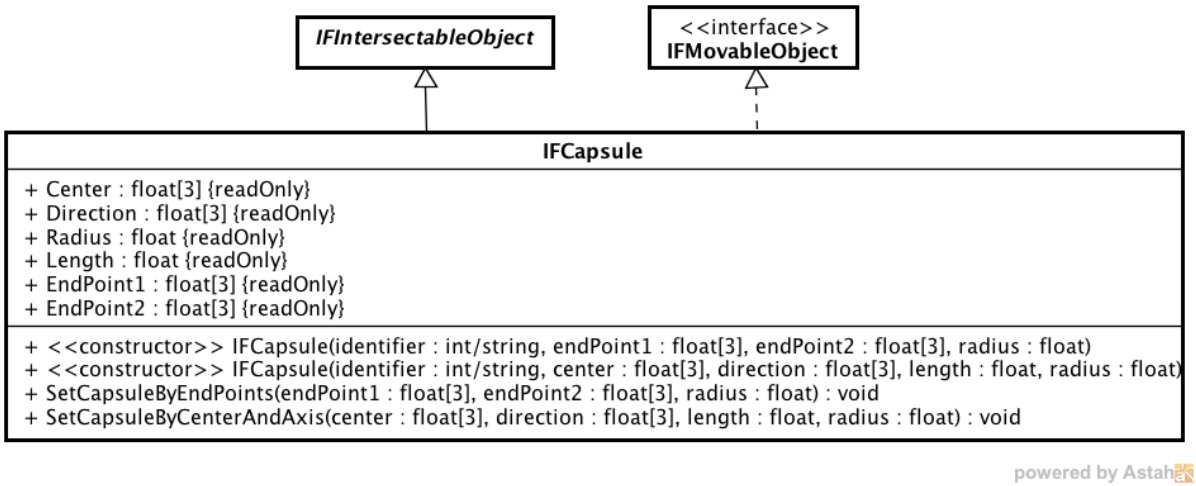


Figura 8.6: La classe *IFSphere*

8.6 La classe IFAABB

Un *axis-aligned bounding box* è un parallelepipedo rettangolo tale che le normali alle sue facce sono parallele agli assi del sistema di riferimento. Un AABB viene costruito a partire dai due punti agli estremi della sua diagonale, oppure specificandone il centro e dimensioni.

È possibile modificare le caratteristiche del box in un secondo momento grazie ai metodi **SetBBoxCoords** e **SetBBoxCenterAndSizes**.

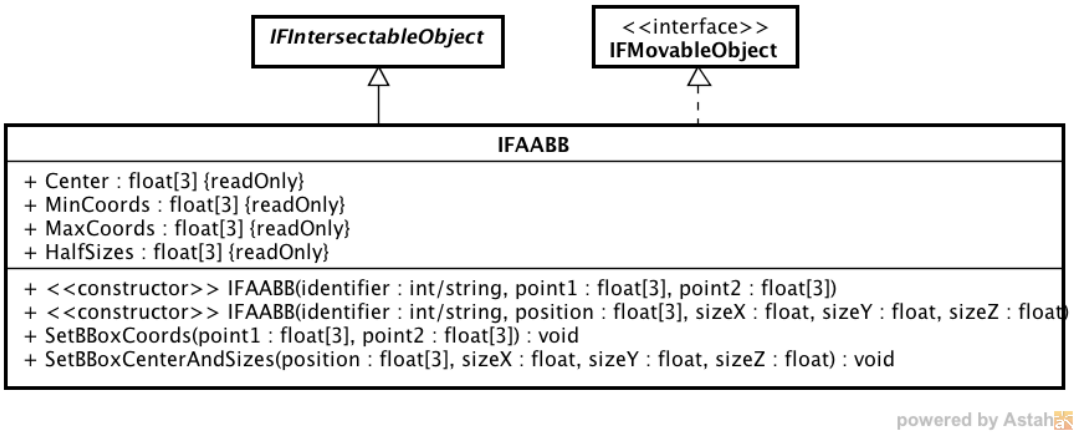


Figura 8.7: La classe *IFAABB*

In realtà, poiché questo è il bounding volume scelto per realizzare il BVH, la classe definisce qualche metodo aggiuntivo a supporto di *IFAABBTree*. È però improbabile che il programmatore abbia la necessità di sfruttarli direttamente, per cui se ne farà cenno solo nella trattazione di *IFAABBTree*. In caso di necessità è comunque possibile ottenere maggiori dettagli su di essi nella documentazione del framework.

8.7 La classe IFOBB

Un *oriented bounding box* è un parallelepipedo rettangolo avente un orientamento arbitrario. Il suo orientamento iniziale viene stabilito al momento della costruzione (come tripla di angoli di Eulero) e può essere modificato in un secondo momento grazie ai metodi dichiarati nell'interfaccia *IFRotatableObject*. Il campo **Axes** contiene un array i cui elementi sono gli assi del sistema di riferimento locale all'oggetto.

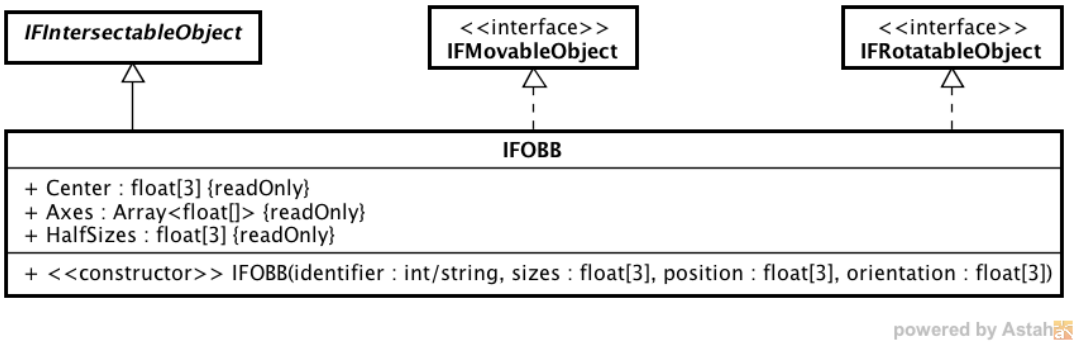


Figura 8.8: La classe *IFOBB*

8.8 La classe IFIntersectableMesh

Questa classe consente di incapsulare una mesh di triangoli[8] al fine di poter applicare ad essa le stesse operazioni previste per le altre primitive geometriche, ed in particolare i test di intersezione descritti in precedenza. Si consiglia comunque di usare mesh composte da un basso numero di triangoli e, quando possibile, usare un bounding volume al suo posto. *IFIntersectableMesh* mette a disposizione il metodo **GetNumberOfTriangles** per conoscere il numero di triangoli di cui è costituita la mesh.

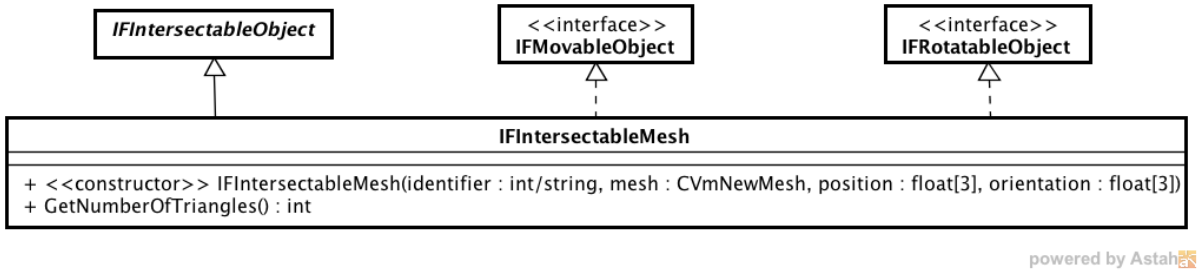


Figura 8.9: La classe *IFIntersectableMesh*

[8] Descritta da un oggetto di tipo CVmNewMesh.

8.9 La classe IFAABBTree

La classe *IFAABBTree* implementa una *bounding volume hierarchy*. Più precisamente, si tratta di un albero binario nel quale ad ogni nodo è associato *axis-aligned bounding box* (istanza di *IFABB*). I nodi foglia rappresentano oggetti di tipo *IFIntersectableObject*, quindi l'AABB del nodo è quello ricavato dall'oggetto. I bounding box associati ai nodi interni hanno invece un volume tale da racchiudere completamente il loro sotto-albero. La scelta di usare l'AABB come *bounding volume* dell'albero è motivata sia dalla semplicità con la quale si riescono a realizzare le operazioni necessarie alla costruzione dell'albero, sia dall'efficienza dei suoi algoritmi per la verifica dell'intersezione.

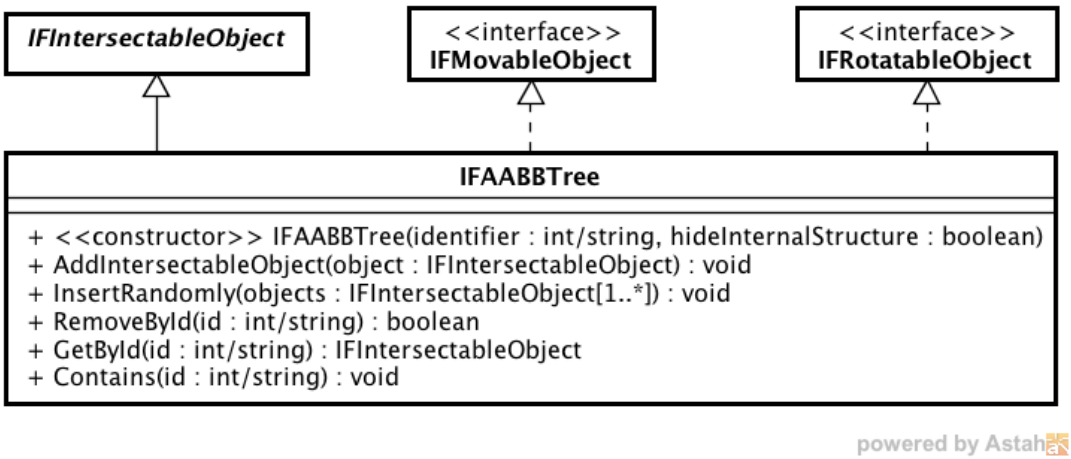


Figura 8.10: La classe *IFAABBTree*

Il costruttore della classe *IFAABBTree* prevede due parametri: un identificatore e un flag booleano usato per nascondere la struttura interna dell'albero. Se questo parametro viene impostato a *true*, i test intersezione forniranno come risultato l'id dell'albero anziché quello di un elemento contenuto in esso. Questo consente di trattare l'albero come una singola primitiva geometrica. Anche se il parametro viene impostato a false, l'identificatore risulta comunque necessario per poter inserire l'albero all'interno di un altro contenitore.

La classe *IFAABBTree* implementa i metodi dichiarati dalle interfacce *IFRotatableObject* e *IFMovableObject*, consentendo così di poter movere e ruotare liberamente l'albero all'interno della scena.

Visita dell'albero

I test di intersezione con la gerarchia consistono in una semplice visita dell'albero nella quale, per ogni nodo incontrato, viene effettuato un test di intersezione con il suo bounding box. Se tale test ha esito negativo la visita di quel sotto-albero termina. In caso contrario

si procede ricorsivamente, testando i sotto-alberi sinistro e destro. Nel caso dei nodi foglia, alla verifica sul bounding box segue il test sull'elemento geometrico associato al nodo (generalmente più costoso).

Di seguito viene riportato lo *pseudo-codice* dell'algoritmo. Il codice si riferisce al solo test di intersezione con un raggio, ma il modo di operare è valido per tutti test di intersezione. Per ragioni di chiarezza si suppone che il risultato del test sia la sola distanza tra l'origine del raggio ed il punto di intersezione (anziché un'istanza di *IFIntersectionResult*). Se il test fallisce viene restituita una distanza "infinita".

```
function AABBTreeNode::IntersectRay(ray)
{
   if(!this.BBox.IntersectRayTest(ray)) return INFINITY;

   var rightInterResult = this.rightChild.IntersectRay(ray);
   var leftInterResult = this.leftChild.IntersectRay(ray);

   return min(rightInterResult, leftInterResult);
}
```

```
function AABBTreeLeaf::IntersectRay(ray)
{
   if(this.BBox.IntersectRayTest(ray))
      return this.IntersectableObject.IntersectRay(ray);
   else return INFINITY;
}
```

Si noti che se il raggio interseca più di un elemento geometrico, il risultato sarà quello di distanza minima. Questo vale anche per il test con una sfera in movimento, mentre nel caso della sfera statica viene restituito l'identificatore del primo elemento per il quale il test dà esito positivo.

Si noti inoltre il suffisso "*Test*" nel nome del metodo usato per testare l'intersezione con il bounding box. Si tratta di una versione ottimizzata dell'algoritmo implementato da *IntersectRay* che restituisce un valore booleano, evitando così di calcolare esplicitamente il punto di intersezione, la normale alla superficie ecc. Nel caso di *CollideWithMovingSphere*, il corrispondente metodo "*Test*" effettua un'ottimizzazione ben più significativa, dovendo però accettare la possibilità che si verifichino falsi positivi.

Il costo di un singolo test di intersezione con la gerarchia dipende principalmente da due fattori:

1. Il bilanciamento dell'albero.

2. Il numero di oggetti intersecati. Questo a sua volta dipende dalla distribuzione nello spazio degli oggetti contenuti nella gerarchia[9] e dalle caratteristiche della primitiva geometrica oggetto del test (sfera o raggio) in relazione alla disposizione degli elementi della gerarchia[10].

Escludendo i casi in cui il test fallisce, il caso ottimo si ha quando l'albero è completamente bilanciato e la primitiva interseca un solo oggetto. In tal caso la complessità del test è logaritmica nel numero di oggetti contenuti nella gerarchia ($\mathcal{O}(\log_2 n)$). Il caso pessimo, invece, si verifica quando vengono visitati tutti i nodi interni dell'albero e vengono testati tutti gli oggetti in esso contenuti. Questo può accadere quando l'albero è completamente sbilanciato oppure quando il raggio interseca tutti gli oggetti. La complessità in questo caso è dunque lineare. Nel caso medio questo tipo di strutture dati consente di ottenere una complessità sublineare.

Costruzione dell'albero

Esistono varie strategie per la costruzione di un BVH, ognuna delle quali presenta caratteristiche diverse in termini di costo computazionale, struttura dell'albero prodotto e difficoltà nell'implementazione dell'algoritmo. La maggior parte di tali algoritmi possono essere classificati in tre famiglie, in funzione della strategia di costruzione usata [11, 28]:

- **Top-down**: Dato un insieme iniziale di oggetti, questi algoritmi partizionano l'insieme in due (o più) sottoinsiemi , calcolano un bounding volume che li racchiuda e proseguono ricorsivamente il partizionamento. Il nodo radice è costituito da un bounding volume che ingloba l'intero insieme iniziale. La loro implementazione è abbastanza semplice, ma gli alberi prodotti non sempre risultano i migliori possibili.

- **Bottom-up**: Si parte da un insieme di oggetti che costituiscono le foglie dell'albero e, ad ogni passo, si crea un nuovo livello della gerarchia raggruppando gli elementi del livello sottostante. Gli algoritmi appartenenti a questa categoria sono i più complicati da realizzare, ma tendenzialmente producono i risultati migliori.

- **Costruzione incrementale**: La costruzione avviene in maniera incrementale, aggiungendo all'albero un nodo alla volta. A differenza delle altre due strategie, per le quali è necessario che tutti gli elementi che andranno a comporre l'albero siano noti al momento della creazione, questa classe di algoritmi consente di aggiungere e rimuovere nodi dall'albero in un qualsiasi momento, rendendone così possibile l'utilizzo in ambienti dinamici. Inoltre, gli alberi prodotti risultano in genere migliori di quelli ottenuti con la strategia *Top-down*.

[9] Che come vedremo influenza la struttura dell'albero e quindi il suo bilanciamento.
[10] Il punto di origine del raggio, la sua lunghezza e direzione ecc.

La classe *IFAABBTree* implementa un algoritmo di costruzione incrementale. L'aggiunta di un nuovo nodo all'albero consiste in una visita dello stesso al fine di individuare il punto di inserimento che minimizza una certa funzione costo. Generalmente si considera come costo dell'inserimento (in una data posizione) l'incremento complessivo del volume dell'albero causato dall'aggiunta del nodo in quel punto. L'effettivo inserimento consiste nella creazione di un nuovo nodo interno a cui viene assegnato come figlio destro il nodo precedentemente situato nel punto scelto, e come figlio sinistro il nodo da inserire. Infine occorre ridimensionare i bounding-box di tutti gli antenati. Segue che il costo di inserimento si ottiene sommando al volume del nuovo padre, l'incremento di volume degli antenati.

Per meglio comprendere l'algoritmo di costruzione, ci serviremo degli esempi illustrati in figura 8.11 dove, per semplicità, l'algoritmo viene simulato in uno spazio bidimensionale. Prendiamo in considerazione la situazione mostrata al punto (a) e supponiamo di voler aggiungere all'albero l'elemento *N1* del punto (b). La visita inizia dal nodo radice *C*, il quale calcola l'incremento di area che si avrebbe inserendo *N1* nella sua posizione[11]. Poiché si tratta del nodo radice (che quindi non ha antenati), il costo è dato dall'area dell'eventuale nuovo nodo padre. Poiché il nuovo nodo padre dovrebbe inglobare completamente sia *C* che *N1*, si ottiene un costo pari a 12 (quadrati). Prima di procedere nell'inserimento, *C* deve valutare se l'inserimento in uno dei suoi sottoalberi risulterebbe più conveniente o meno. A tal fine chiede ai nodi *A* e *B* di valutare il costo di un eventuale inserimento. Il calcolo si ripete allo stesso modo, con la differenza che stavolta è necessario considerare l'incremento subito da *C* a seguitoinserimento di *N1* in uno dei suoi sottoalberi (A = 6). Nell'esempio risulta che il costo si minimizza inserendo *N1* nel sottoalbero radicato in *B*. La procedura procede quindi ricorsivamente, visitando il nodo *B*. Nel caso specifico *B* è un nodo foglia, quindi si limita a creare un nuovo nodo padre (*D*) avente *B* e *N1* come figli ed a restituirne il riferimento come risultato della procedura, così che *C* possa aggiornarsi di conseguenza.

In figura 8.11 (d)-(e) viene invece mostrato, a titolo di esempio, un caso nel quale la posizione di inserimento ottimale risulta essere uno dei nodi interni (la radice).

Si noti infine che questo algoritmo produce alberi nei quali ogni nodo interno ha sempre due figli.

La classe *IFAABBTree* fornisce il metodo **AddIntersectableObject** per inserire un oggetto di tipo *IFIntersectableObject* nell'albero. Di seguito viene riportata sua implementazione[12]. Si noti che:

[11] In figura viene indicato con "A" l'incremento totale dell'area degli antenati, con "P" l'area del nuovo nodo padre e con "C" il costo complessivo dato dalla somma dei due.

[12] In realtà questo codice differisce leggermente dall'originale in quanto, per ragioni di spazio e di chiarezza, alcuni nomi sono stati modificati.

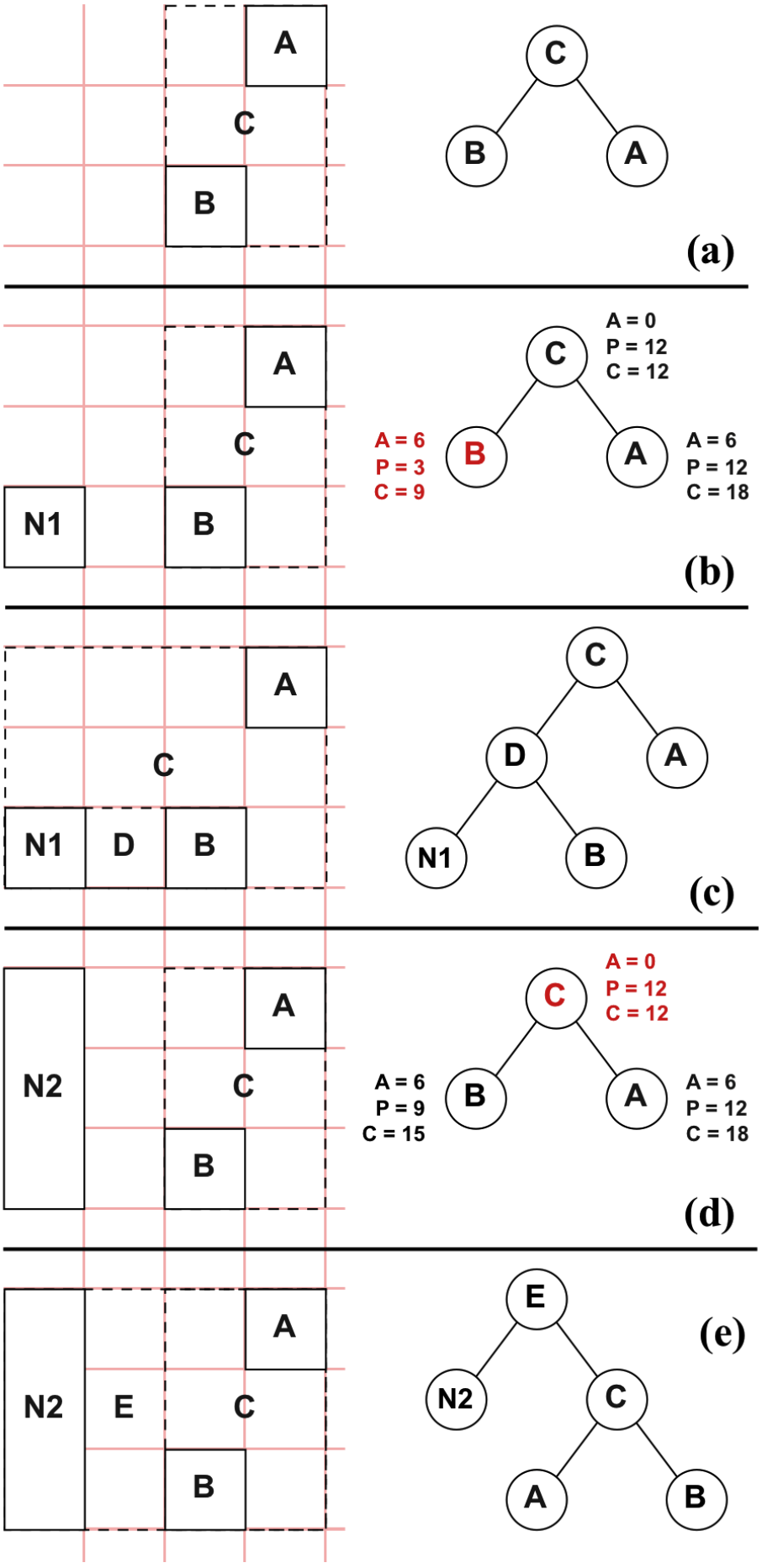


Figura 8.11: Algoritmo per l'inserimento di un nodo nell'albero di bounding-box

- *IFBBoxTreeNode* e *IFBBoxTreeLeaf* sono le classi che rappresentano rispettivamente i nodi interni ed i nodi foglia di *IFAABBTree*.

- Il metodo *EvaluateCost* calcola il costo di inserimento relativamente alla posizione occupata dal nodo oggetto dell'invocazione. I due parametri richiesti sono il nodo da inserire e l'incremento di volume degli antenati. Questo metodo si occupa anche di memorizzare il costo totale ed il nuovo incremento di volume degli antenati (da passare ai figli) rispettivamente nei campi (privati) "*totalCost*" e "*ancestorsIncrease*", in modo che non debbano essere ricalcolati da *AddNode*.

```
function IFAABBTree::AddIntersectableObject(obj)
{
   var newNode = IFBBoxTreeLeaf(obj);
   if(this.root == NULL)
      this.root = newNode;
   else
   {
      this.root.EvaluateCost(newNode, 0);
      this.root = this.root.AddNode(newNode);
   }
}
```

```
function IFBBoxTreeNode::EvaluateCost(newNode, ancestorsVolumeIncrease)
{
   var newVolumeToContain = BBox.GetVolumeIncrease(newNode.BBox);
   this.ancestorsIncrease = newVolumeToContain + ancestorsVolumeIncrease;
   this.totalCost = this.ancestorsIncrease + BBox.Volume;
   return this.totalCost;
}
```

```
function IFBBoxTreeNode::AddNode(newNode)
{
   var leftCost = leftChild.EvaluateCost(newNode, this.ancestorsIncrease);
   var rightCost = rightChild.EvaluateCost(newNode, this.ancestorsIncrease)
      ;
   if( this.totalCost < leftCost && this.totalCost < rightCost )
   {
      var newParentBBox = BBox.Union(newNode.BBox);
      var newParent = IFBBoxTreeNode(newParentBBox, this, newNode);
      return newParent;
   }
   else
   {
      BBox.ExpandToContain(newNode.BBox);
      if(leftCost < rightCost)
         this.leftChild = this.leftChild.AddNode(newNode);
```

```
        else
            this.rightChild = this.rightChild.AddNode(newNode);

        return this;
    }
}
```

```
function IFBBoxTreeLeaf::AddNode(newNode)
{
    var parentBox = this.BBox.Union(newNode.BBox);
    var newParent = IFBBoxTreeNode(parentBox, this, newNode);
    return newParent;
}
```

Una delle caratteristiche di questo algoritmo da tenere bene in considerazione consiste nel fatto che la struttura finale dell'albero dipende, oltre che dalla distribuzione nello spazio degli oggetti che lo compongono, anche dal loro ordine di inserimento. Per minimizzare la probabilità di ottenere alberi poco bilanciati, è consigliabile inserire gli oggetti nell'albero in ordine casuale. Di questo si occupa il metodo **InsertRandomly**, che richiede come parametro un array contenente gli oggetti da aggiungere. L'uso di questo metodo è consigliato ogni volta che vi è la necessità di inserire un numero significativo di nuovi oggetti, e non solo per la costruzione iniziale.

8.10 La classe IFIntersectableObjectsSet

Rappresenta un insieme di *IFIntersectableObject*. Oltre ai classici metodi di ogni contenitore, questa classe fornisce una variante dei metodi dichiarati dalle interfacce *IFRotatableObject* e *IFMovableObject* che consentono di modificare la posizione o l'orientamento di un oggetto contenuto nell'insieme. Tali metodi, uniti alla proprietà di poter reperire un oggetto dell'insieme in tempo costante, rendono questa classe un buon contenitore per oggetti dinamici. Ovviamente è necessario mantenere bassa la cardinalità dell'insieme, in quanto la complessità dei test di intersezione è lineare nel numero di oggetti in esso contenuti. Tuttavia nell'ambito applicativo di riferimento (fruizione di contenuti culturali) difficilmente il numero di oggetti dinamici è elevato.

Questo tipo di contenitore risulta utile anche in alcune situazione particolari, nelle quali l'uso di un BVH comporterebbe un peggioramento delle prestazioni. Si pensi ad esempio al caso di quattro parallelepipedi disposti lungo i lati di un quadrato.

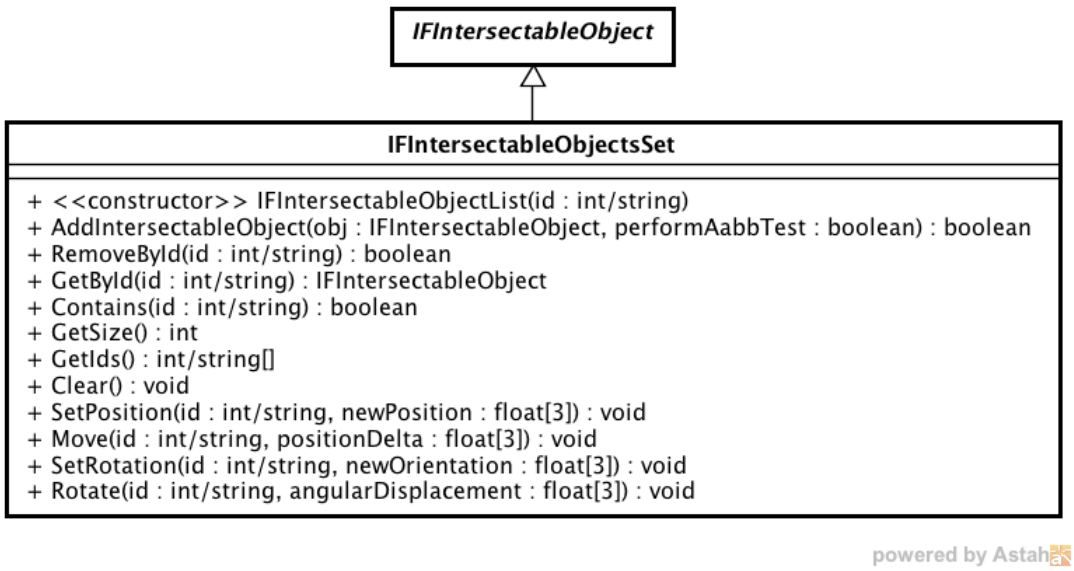


Figura 8.12: La classe *IFIntersectableObjectsSet*

Infine, si noti che il metodo **AddIntersectableObject** richiede un secondo parametro booleano, chiamato "*performAabbTest*". Se tale parametro viene settato a true, al momento dell'inserimento viene creato un AABB per il nuovo oggetto ed i test di intersezione verranno eseguiti prima su di esso. A seconda del tipo di elemento inserito, questo può portare ad un miglioramento (ad es. nel caso di mesh) od un peggioramento (ad es. per le sfere) delle prestazioni. Per tale ragione la scelta del valore da assegnare deve essere valutata con attenzione. Essendo un parametro opzionale, è anche possibile ometterlo. Nel qual caso verrà considerato *false*.

Parte III

Risultati sperimentali

Capitolo 9

Prestazioni della gerarchia di bounding box

Per verificare le effettive prestazioni della gerarchia di bounding-box (descritta nel paragrafo 8.9) e confrontarle con gli strumenti messi a disposizione da XVR, è stata realizzata una semplice simulazione nella quale viene creata una scena in maniera pseudo-casuale e viene misurato il tempo necessario all'esecuzione dei test di intersezione tra un raggio e gli oggetti presenti nella scena.

L'uso di una gerarchia di bounding-box consente di ridurre drasticamente il costo di un test di intersezione con gli elementi in essa contenuti, a patto però che l'albero sia sufficientemente bilanciato. L'algoritmo di costruzione implementato dalla classe *IFAABBTree* minimizza, ad ogni inserimento, l'incremento di volume dei bounding box associati a nodi dell'albero. Questo algoritmo però non garantisce il bilanciamento dell'albero, in quanto la struttura dello stesso dipende sia dalla disposizione degli oggetti nello spazio, sia dal loro ordine di inserimento. Per cercare evitare uno sbilanciamento troppo pronunciato, la classe *IFAABBTree* fornisce un metodo specifico per la costruzione iniziale dell'albero a partire da un insieme di oggetti (*InsertRandomly*), che provvede ad inserire gli elementi dell'insieme in ordine casuale. In conclusione, quando l'albero risulta completamente bilanciato (caso ottimo) il test di intersezione ha complessità $\mathcal{O}(\log_2 n)$ (dove n è il numero di oggetti sui quali viene eseguito il test)[1]. Sono però possibili casi in cui l'albero è completamente sbilanciato (caso pessimo), con conseguente costo lineare nel numero di oggetti. Scopo della simulazione è quindi quello di verificare se, nel caso medio, il test di intersezione ha complessità sub-lineare.

La realizzazione di *IFAABBTree* è stata motivata dal fatto che XVR non espone alcuna *struttura dati spaziale*[2] concepita per eseguire efficientemente test di questo genere. Nella

[1] Nell'ipotesi che venga intersecato uno ed un solo oggetto.

[2] Una struttura dati spaziale è una struttura dati usata per organizzare la geometria in uno spazio

pratica si ricorre quindi ad un espediente: si crea una porzione di scene graph e si verifica l'intersezione del raggio con gli elementi che lo compongono, invocando il metodo *IsColliding* sulla radice. I nodi interni dello scene graph sono oggetti di tipo *CVmObj*, la cui implementazione del metodo *IsColliding* si limita a propagare la richiesta a tutti i nodi figli. È quindi lecito aspettarsi che la complessità del test sia lineare. In ogni caso, per avere un termine di paragone, è stata svolta anche una simulazione con un albero di *CVmObj*.

9.1 Svolgimento della prova

La prova è costituita da tre simulazioni, una per ogni tipo di gerarchia impiegata:

1. *IFAABBTree*, usando un algoritmo di verifica dell'intersezione con i bounding box dell'albero implementato in linguaggio S3D.
2. *IFAABBTree*, usando un algoritmo di verifica dell'intersezione con i bounding box dell'albero implementato in linguaggio C.
3. Albero di *CVmObj*. L'albero viene generato automaticamente dalle istanze di *CVmObj* man mano che vengono aggiunti nuovi nodi.

In ogni simulazione vengono generati 50 scenari, ognuno dei quali caratterizzato da una differente scena tridimensionale o da una differente gerarchia. In particolare, dato un insieme di n mesh triangolari di forma cubica (ciascuna composta da dodici triangoli), ciascuna scena viene creata disponendo casualmente le mesh su di una griglia di dimensioni 100 x 100 x 100. Nelle simulazioni relative a *IFAABBTree* vengono generate 5 scene distinte e per ognuna di esse si costruiscono 10 gerarchie di bounding box. Poiché l'inserimento avviene in ordine casuale e poiché la struttura dell'albero dipende da tale ordine, vi è una buona probabilità di ottenere alberi differenti tra loro[3], consentendo così di verificare l'impatto di questo aspetto sulle prestazioni. Nella simulazione relativa all'albero di *CVmObj*, vengono invece create 50 scene distinte[4].

Per ogni scenario si effettuano infine 20 misurazioni[5]. Ciascuna misurazione consiste nel ricavare il tempo totale necessario ad eseguire il test di intersezione con tutte le mesh della scena. È importane notare che il test di intersezione con la mesh viene svolto dal metodo *IsColliding* (fornito da XVR), ed è quindi indipendente dal tipo di gerarchia usato.

n-dimensionale [27] (nel nostro caso tridimensionale). Esempi tipici sono le BVH, gli alberi BSP, e gli *octree*.

[3] Per valori di n sufficientemente grandi.

[4] In caso contrario la struttura dell'albero risulterebbe sempre la stessa.

[5] Per evitare che eventuali fattori esterni possano interferire con i risultati della prova

Inoltre, il raggio con il quale eseguire il test viene generato in maniera tale da intersecare la sola mesh considerata.

La prova deve essere poi ripetuta per valori crescenti di n.

Tutte le prove sono state eseguite su di un calcolatore avente le seguenti caratteristiche:

Processore: Intel Pentium 4 - 3.4 GHz

Memoria principale: 1 GB

Sistema operativo: Windows Vista Business 32bit

Scheda grafica: NVIDIA GeForce 6800

9.2 Elaborazione dei dati raccolti

I tempi ottenuti dalle misurazioni sono stati divisi per n, in modo da ottenere il tempo medio necessario ad eseguire un singolo test di intersezione.

I valori relativi ad ogni simulazione sono stati poi organizzati in ordine crescente e da questo insieme è stato scartato il 10% delle misurazioni che presentava valori più bassi ed il 10% che presentava valori più alti.

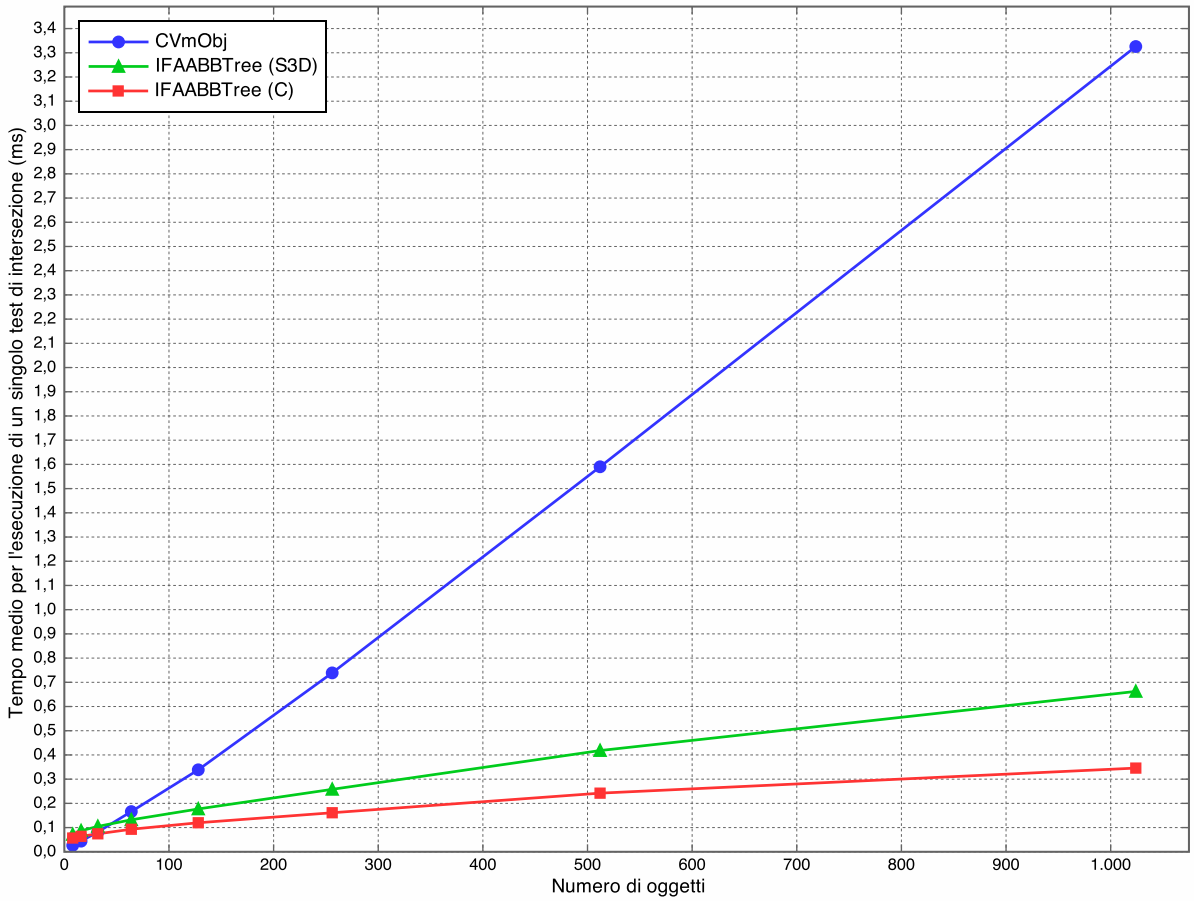
Dei valori rimanenti è stata calcolata la media, ottenendo così un singolo valore medio per ogni simulazione.

9.3 Risultati ottenuti

I risultati ottenuti sono riportati nella tabella 9.1.

La prova conferma le ipotesi teoriche, in quanto risulta che, nel caso medio, il test di intersezione tra un raggio ed una gerarchia di bounding-box ha complessità sub-lineare. La complessità del medesimo test sull'albero di *CVmObj* si è dimostrata lineare, secondo le aspettative.

È interessante osservare che le prestazioni ottenute con l'implementazione in linguaggio S3D (poi compilato in bytecode ed eseguito su di una virtual machine) non si differenziano molto dalle prestazioni dell'implementazione in C. Questo è dovuto al fatto che l'altezza dell'albero è sempre molto inferiore al numero di nodi foglia, quindi il numero di test di intersezione effettivamente eseguiti è troppo basso per evidenziare differenze significative. Si può quindi concludere che, sfruttando la gerarchia di bounding-box, anche l'implementazione in linguaggio S3D risulta più che adeguata allo scopo.



Nodi	CVmObj (ms)	IFAABBTree - S3D (ms)	IFAABBTree - C (ms)
8	0.026565	0.071598	0.056817
16	0.043979	0.086926	0.063923
32	0.081754	0.104843	0.074289
64	0.166097	0.132145	0.093125
128	0.338625	0.176960	0.119946
256	0.738875	0.257957	0.161275
512	1.590146	0.418170	0.242180
1024	3.325864	0.662370	0.345706

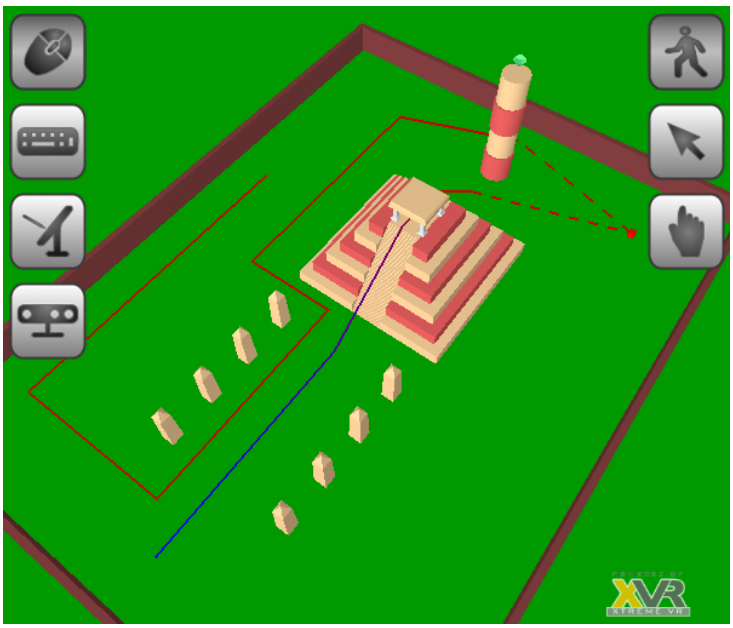
Tabella 9.1: Tempo medio impiegato dall'esecuzione di un singolo test di intersezione tra un raggio ed una gerarchia di oggetti

Capitolo 10

Test di usabilità

Al fine di avere un feedback sull'efficacia delle metafore e delle tecniche di interazione implementate, e poterne fare un confronto, sono stati svolti alcuni test sull'usabilità delle stesse.

Agli utenti è stato chiesto di navigare all'interno dell'ambiente virtuale lungo un percorso prestabilito, seguendo degli opportuni indicatori visivi disegnati all'interno della scena. A circa metà percorso è stato inserito un oggetto speciale, che gli utenti dovevano selezionare. L'applicazione provvedeva a cronometrare il tempo impiegato nello svolgimento di tali compiti. Il passaggio dalla modalità di navigazione a quella di selezione era reso possibile da un'apposita interfaccia grafica costituita da alcuni bottoni.



Il percorso è stato ripetuto più volte da ogni utente, utilizzando ogni volta una diversa metafora di navigazione o di selezione ed un diverso dispositivo di input. In particolare, data la natura del tragitto, le metafore di navigazione scelte sono state il *Flying Vehicle*,

il *Walking* e lo *Speed-coupled Flying*. Quanto alla selezione, sono state impiegate le uniche due metafore disponibili: la *Ray selection* e la *Cursor selection*. I dispositivi di input utilizzati con ciascuna metafora sono[1]:

- *Flying Vehicle*: mouse, tastiera e mouse, touchscreen, Kinect
- *Walking*: mouse, tastiera e mouse, touchscreen, Kinect
- *Speed-coupled Flying*: mouse, tastiera e mouse, Kinect
- *Ray selection*: mouse, touchscreen, Kinect
- *Cursor selection*: mouse, Kinect

Le prove fin qui descritte sono state svolte da quattro utenti, che al termine hanno rilasciato una breve intervista in merito alle loro impressioni di utilizzo. I dati, relativi ai tempi impiegati dagli utenti per lo svolgimento dei vari task, sono stati organizzati in funzione della metafora e del dispositivo utilizzato, e per ciascuna categoria è stato ricavato il valor medio. I risultati così ottenuti sono stati infine riportati su grafico, in modo da agevolare la loro analisi.

Viste la dimensione del campione e le caratteristiche del test, i risultati ottenuti non possono avere valenza statistica. D'altra parte non era questo lo scopo della prova, bensì quello di ottenere una prima valutazione qualitativa dell'usabilità delle metafore e delle tecniche di interazione.

10.1 Risultati

I tempi medi impiegati dagli utenti per completare la navigazione della scena sono riportati in fig. 10.1 e 10.2, organizzati rispettivamente in funzione della metafora e del dispositivo usato. I risultati riguardanti la selezione sono invece riportati in fig. 10.3.

Come era prevedibile, i risultati migliori sono stati ottenuti utilizzando il mouse, da solo e in combinazione con la tastiera. Fissati questi dispositivi, le prestazioni ottenute con le varie metafore sono del tutto comparabili, anche se si può notare una piccola differenza a favore del *Walking*. Questo è dovuto al fatto che la metafora vincola la posizione della camera al terreno, rendendo più semplice seguire il tracciato.

Il Kinect si è dimostrato il dispositivo di input più difficile da controllare, come confermano i relativi tempi medi di completamento del percorso. Questo dispositivo richiede infatti un certo allenamento prima di poter essere usato agevolmente. Inoltre, la sua scarsa sensibilità è causa di frequenti errori di rilevamento delle mani, che provocano l'interruzione

[1] In accordo ai controller definiti nel framework.

delle azioni in corso e disorientano gli utenti meno esperti. Con una certa pratica si riesce comunque a prevenire tali comportamenti.

Valutando soltanto i risultati ottenuti, il touchscreen e la relativa tecnica di interazione si sono rivelati sufficientemente efficaci. Certo, i tempi medi di completamento non sono buoni come quelli ottenuti da tastiera e mouse, ma si deve anche considerare che questo genere di dispositivo mal si adatta al controllo della navigazione in una scena tridimensionale, se non coadiuvato da una opportuna interfaccia grafica (che però non ne sfrutta le capacità).

Quanto alle metafore di selezione, i risultati ottenuti evidenziano ancora una volta i pessimi risultati ottenuti con il Kinect. In questo caso, però, una parte delle cause è da attribuirsi alle tecniche di interazione implementate. Infatti si è cercato di rendere l'interazione il più naturale possibile, associando la direzione del raggio e la posizione del cursore rispettivamente con la direzione del braccio e la posizione della mano. Tuttavia il dispositivo non riesce a fornire un tracciamento sufficientemente stabile, rendendo così difficile "centrare" l'elemento da selezionare.

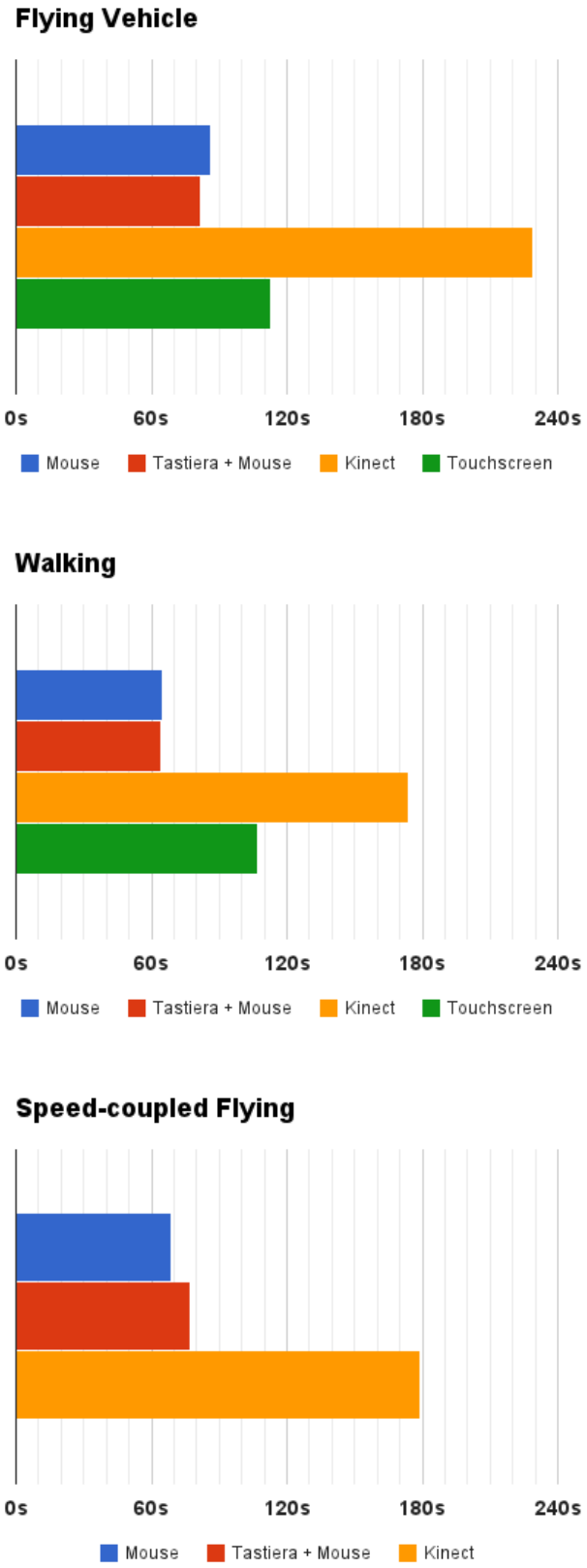


Figura 10.1: Tempo medio impiegato per completare il percorso (solo navigazione)

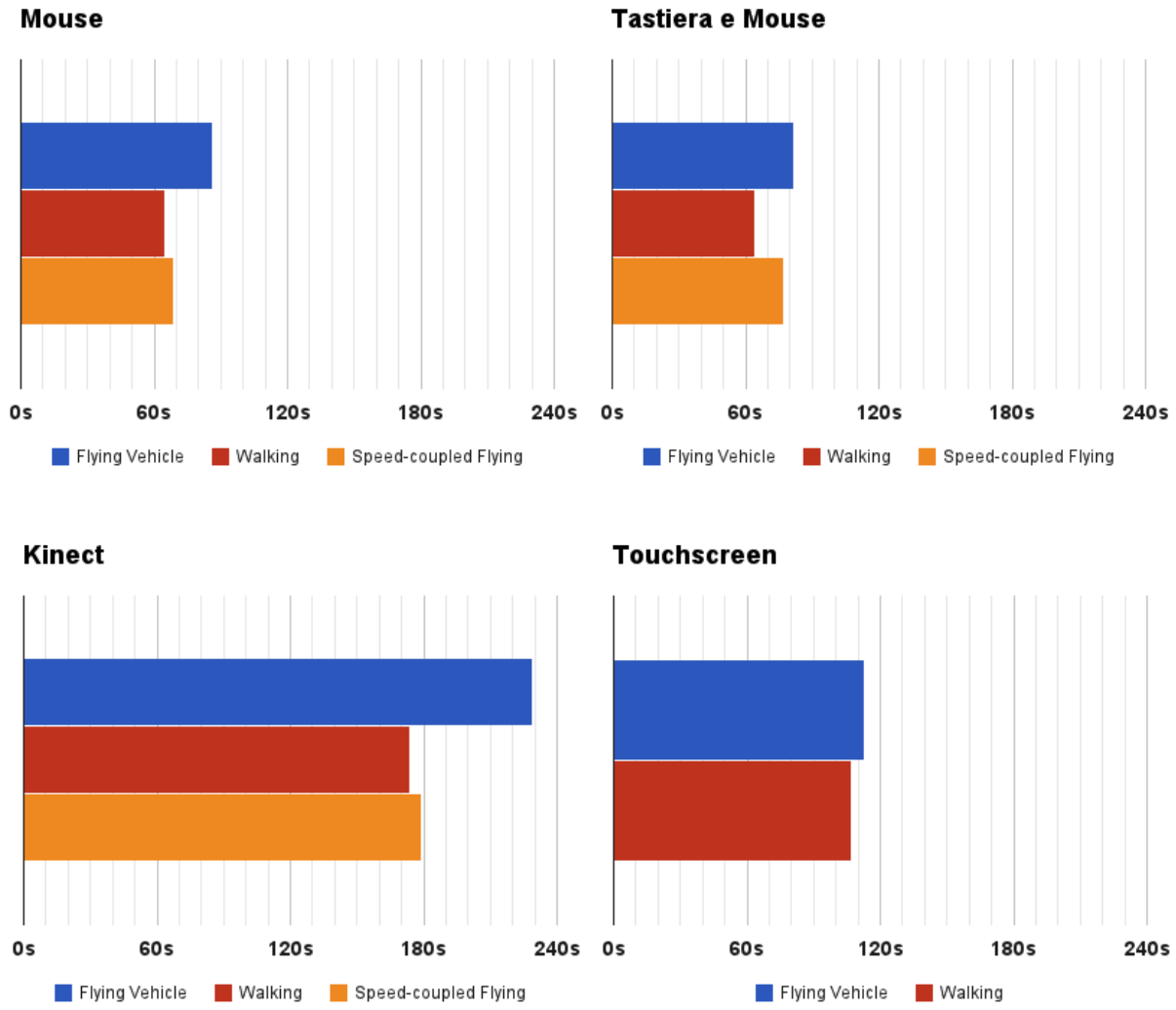


Figura 10.2: Tempo medio impiegato per completare il percorso (solo navigazione)

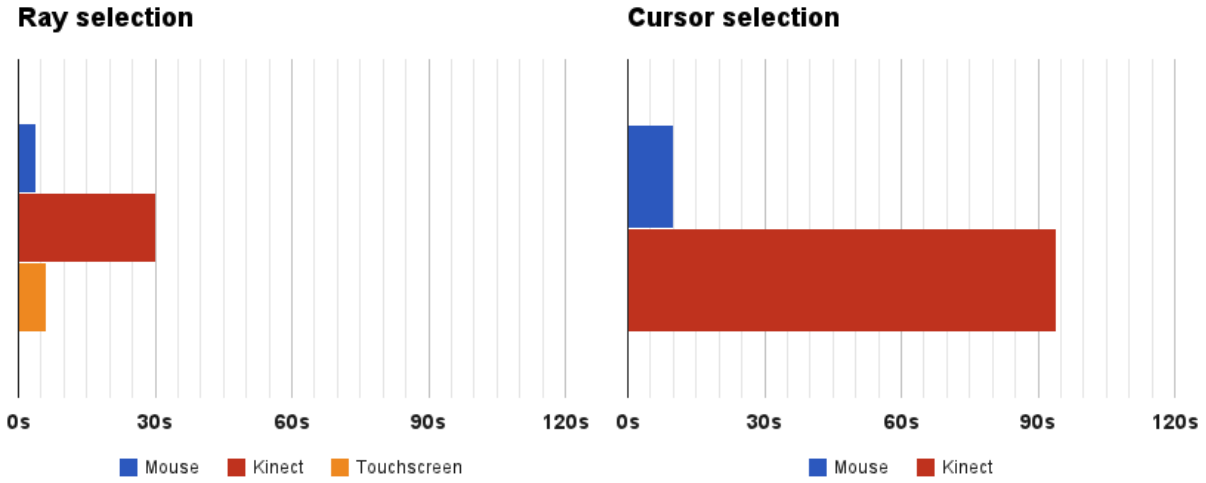


Figura 10.3: Tempo medio impiegato per selezionare un elemento della scena

10.2 Feedback degli utenti

Vediamo infine le considerazioni espresse dagli utenti in una breve intervista rilasciata al termine della prova .

Quanto alle metafore di navigazione, quasi tutti gli utenti hanno preferito il *Walking* al *Flying Vehicle*, in quanto la funzionalità di *terrain following* consentiva loro di seguire il tracciato con minore difficoltà. Scarso successo ha invece avuto lo *Speed-coupled Flying*, la cui modalità di volo è stata da tutti ignorata dopo il primo tentativo di utilizzo. Probabilmente questo comportamento è stato indotto dalle dimensioni contenute della scena, che non hanno consentito di sfruttare efficacemente questa metafora.

Tra le metafore di selezione la Ray selection è stata la più apprezzata, quanto meno quando abbinata a mouse e touchscreen. La selezione via Kinect è stata giudicata problematica per entrambe le metafore. Solo un utente ha espresso la sua preferenza per la cursor selection, che in effetti dovrebbe essere la più indicata per questo genere di dispositivi, in quanto tipica degli ambienti immersivi. Questo però non trova riscontro nei risultati ottenuti.

Più in generale, il Kinect è stato da tutti considerato molto divertente da usare, ma solo dopo un certo periodo di addestramento. Tutti concordano anche sul fatto che questo dispositivo è decisamente più usabile quando abbinato alla metafora *Walking*. Infatti, i minori gradi di libertà offerti dalla metafora hanno compensato in parte le difficoltà di controllo. A tal proposito, il "Kinect Drawer" (si veda il paragrafo 4.5.1) il si è dimostrato di fondamentale aiuto ed è stato apprezzato esplicitamente dalla metà degli utenti.

Quanto al touchscreen, quasi tutti i partecipanti hanno trovato difficoltà nel padroneggiare le tecniche di interazione per la navigazione basate su stati (si veda pagina 121). Alcuni hanno però riconosciuto che, data la natura del dispositivo e della metafora, difficilmente si sarebbe potuto pensare a tecniche di interazione molto differenti.

Non sono stati espressi commenti particolari su mouse e tastiera, in quanto considerati da tutti il termine di paragone con il quale confrontare altri dispositivi.

Conclusioni

L'obiettivo di questa tesi era lo sviluppo di un framework per semplificare l'implementazione dell'interazione utente nelle applicazioni XVR, supportando al contempo dispositivi di input di natura diversa. Esso fornisce un insieme di metafore e tecniche di interazione già pronte all'uso e sufficientemente configurabili.

Il framework non si limita però a questo, bensì introduce un livello di astrazione sopra XVR che uniforma e semplifica la gestione dell'input rispetto a quanto avviene nei programmi XVR tradizionali. La scelta di operare una netta distinzione tra metafore di interazione, tecniche di interazione e fonti di eventi di input, rende possibile l'aggiunta o la sostituzione di alcune di queste componenti senza la necessità di apportare modifiche alle altre.

Le metafore presenti sono state scelte tra quelle più usate nelle applicazioni per la comunicazione culturale, ma la loro generalità ne rende possibile l'utilizzo anche in altri ambiti applicativi. Esse presentano un certo grado di configurabilità e supportano funzionalità quali *collision detection* e *terrain following* della camera. A tal fine è stata definita una libreria contenente gli algoritmi e le strutture dati necessarie all'esecuzione dei test di intersezione tra primitive geometriche.

Quanto ai dispositivi di input, l'attenzione è stata rivolta principalmente ai sistemi desktop. Attualmente vengono supportati il mouse, la tastiera, il touchscreen e, in via sperimentale, il Microsoft Kinect. È comunque possibile aggiungere il supporto a nuovi dispositivi (anche di diversa natura), senza che questo comporti la modifica delle altre componenti del framework.

Infine è stata definita una libreria minimale per la realizzazione di semplici interfacce grafiche bidimensionali. Anche in questo caso si è cercato di operare una distinzione tra i componenti grafici e le tecniche di interazione che ne consentono il controllo, al fine di supportare dispositivi di input di natura diversa.

L'API è stata progettata cercando di minimizzare la quantità di codice necessaria all'utilizzo delle varie funzionalità offerte, senza però trascurare i requisiti di semplicità e uniformità. I primi feedback ottenuti dai programmatori PERCRO che hanno avuto modo

di provare il framework (anche se in modo parziale), sono stati positivi e di apprezzamento per il lavoro svolto.

Sviluppi futuri

Lo sviluppo di un framework è potenzialmente illimitato. Ci saranno infatti sempre nuove funzionalità da aggiungere o migliorare.

Nel caso specifico, possono essere definite nuove metafore (soprattutto in relazione ad altri ambiti applicativi) e può essere aggiunto il supporto ad altri tipi di dispositivi, quali quelli tipici dei sistemi immersivi (interfacce aptiche, sistemi di tracking ecc.). Inoltre le tecniche di interazione presenti possono essere perfezionate, tenendo anche conto dei risultati ottenuti nei test di usabilità e mettendo a punto prove più mirate.

Alcuni aspetti del modulo per la realizzazione di interfacce grafiche possono essere certamente migliorati in futuro. Inoltre la collezione dei controlli grafici dovrebbe essere arricchita, così come l'insieme dei dispositivi supportati. Non dovrebbe comportare grandi difficoltà anche l'aggiunta di componenti grafici tridimensionali.

Sarebbe anche interessante realizzare un editor visuale che consenta lo sviluppo di applicazioni attraverso la composizione di elementi grafici associati alle varie metafore, tecniche e dispositivi.

Infine è attualmente in fase di sviluppo un modulo rivolto alle applicazioni "legacy", che consenta con poco sforzo di integrare le funzionalità offerte dal framework nei programmi preesistenti.

Bibliografia

[1] R. Ajaj, F. Vernier, and C. Jacquemin. Navigation modes for combined table/-screen 3d scene rendering. In *Proceedings of the ACM International Conference on Interactive Tabletops and Surfaces*, pages 141–148. ACM, 2009.

[2] Giovanni Avveduto. Studio, progettazione e realizzazione di un'interfaccia per l'interazione naturale in ambienti virtuali immersivi. Master's thesis, Università di Pisa, 2012.

[3] J. De Boeck. Common tasks and interaction metaphors. Internal deliverable of ENACTIVE NoE, 2005.

[4] D.A. Bowman and L.F. Hodges. An evaluation of techniques for grabbing and manipulating remote objects in immersive virtual environments. In *Proceedings of the 1997 symposium on Interactive 3D graphics*, pages 35–ff. ACM, 1997.

[5] F. Buschman, R. Meunier, H. Rohnert, P. Sommerlab, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns, Vol. 1.* John Wiley and Sons, 1996.

[6] M. Carrozzino, F. Tecchia, S. Bacinelli, C. Cappelletti, and M. Bergamasco. Lowering the development time of multimodal interactive application: the real-life experience of the xvr project. In *Proceedings of the 2005 ACM SIGCHI International Conference on Advances in computer entertainment technology*, pages 270–273. ACM, 2005.

[7] J. De Boeck and K. Coninx. Haptic camera manipulation: extending the camera in hand metaphor. 2002.

[8] J. De Boeck, E. Cuppens, T. De Weyer, C. Raymaekers, and K. Coninx. Multisensory interaction metaphors with haptics and proprioception in virtual environments. In *Proceedings of the third Nordic conference on Human-computer interaction*, pages 189–197. ACM, 2004.

[9] F. Dunn and I. Parberry. *3D math primer for graphics and game development.* AK Peters Ltd, 2011.

[10] D.H. Eberly. *3D game engine design: a practical approach to real-time computer graphics*. Morgan Kaufmann Publishers, 1 edition, 2000.

[11] C. Ericson. *Real-time collision detection*. Morgan Kaufmann, 2005.

[12] K. Fauerby. Improved collision detection and response. *Peroxide Entertainment*, 2003.

[13] A. Forsberg, K. Herndon, and R. Zeleznik. Aperture based selection for immersive virtual environments. In *Proceedings of the 9th annual ACM symposium on User interface software and technology*, pages 95–96. ACM, 1996.

[14] Jorge Franchi. Virtual reality: An overview. *TechTrends*, 39:23–26, 1994. 10.1007/BF02763870.

[15] J.L. Gabbard. *A taxonomy of usability characteristics in virtual environments*. PhD thesis, Virginia Polytechnic Institute and State University, 1997.

[16] E. Gamma. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[17] M. Hachet, F. Decle, S. Knodel, and P. Guitton. Navidget for easy 3d camera positioning from 2d inputs. In *3D User Interfaces, 2008. 3DUI 2008. IEEE Symposium on*, pages 83–89. IEEE, 2008.

[18] M. Hachet, F. Decle, S. Knödel, and P. Guitton. Navidget for 3d interaction: Camera positioning and further uses. *International Journal of Human-Computer Studies*, 67(3):225–236, 2009.

[19] V. Interrante, B. Ries, and L. Anderson. Seven league boots: A new metaphor for augmented locomotion through moderately large scale immersive virtual environments. In *3D User Interfaces, 2007. 3DUI'07. IEEE Symposium on*. IEEE, 2007.

[20] S. Knödel, M. Hachet, and P. Guitton. Navidget for immersive virtual environments. In *Proceedings of the 2008 ACM symposium on Virtual reality software and technology*, pages 47–50. ACM, 2008.

[21] D.R. Koller, M.R. Mine, and S.E. Hudson. Head-tracked orbital viewing: an interaction technique for immersive virtual environments. In *Proceedings of the 9th annual ACM symposium on User interface software and technology*, pages 81–82. ACM, 1996.

[22] E. Kruijff. *Unconventional 3D user interfaces for virtual environments*. PhD thesis, PhD thesis, Graz University of Technology, 2006.

[23] C. Larman. *Applying UML and patterns: an introduction to object-oriented analysis and design and iterative development*. Prentice Hall PTR, 3rd edition, 2004.

[24] J.D. Mackinlay, S.K. Card, and G.G. Robertson. Rapid controlled movement through a virtual 3d workspace. In *ACM SIGGRAPH Computer Graphics*, volume 24, pages 171–176. ACM, 1990.

[25] M. Mine. Virtual environment interaction techniques. *UNC Chapel Hill Computer Science Technical Report TR95-018*, pages 507248–2, 1995.

[26] M. Mine. Working in a virtual world: Interaction techniques used in the chapel hill immersive modeling program. *University of North Carolina*, 1996.

[27] T. Möller, E. Haines, and N. Hoffman. *Real-time rendering*. AK Peters Ltd, 2008.

[28] S.M. Omohundro. *Five balltree construction algorithms*. International Computer Science Institute, 1989.

[29] R. Pausch, T. Burnette, D. Brockway, and M.E. Weiblen. Navigation and locomotion in virtual worlds via flight into hand-held miniatures. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 399–400. ACM, 1995.

[30] J.S. Pierce, A.S. Forsberg, M.J. Conway, S. Hong, R.C. Zeleznik, and M.R. Mine. Image plane interaction techniques in 3d immersive environments. In *Proceedings of the 1997 symposium on Interactive 3D graphics*, pages 39–ff. ACM, 1997.

[31] J.S. Pierce, B.C. Stearns, and R. Pausch. Voodoo dolls: seamless interaction at multiple scales in virtual environments. In *Proceedings of the 1999 symposium on Interactive 3D graphics*, pages 141–145. ACM, 1999.

[32] I. Poupyrev, T. Ichikawa, S. Weghorst, and M. Billinghurst. Egocentric object manipulation in virtual environments: empirical evaluation of interaction techniques. In *Computer Graphics Forum*, volume 17, pages 41–52. Wiley Online Library, 1998.

[33] Ivan Poupyrev, Mark Billinghurst, Suzanne Weghorst, and Tadao Ichikawa. The go-go interaction technique: Non-linear mapping for direct manipulation in vr. pages 79–80, 1996.

[34] F. Steinicke, T. Ropinski, and K. Hinrichs. Object selection in virtual environments using an improved virtual pointer metaphor. *Computer Vision and Graphics*, pages 320–326, 2006.

[35] R. Stoakley, M.J. Conway, and R. Pausch. Virtual reality on a wim: interactive worlds in miniature. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 265–272. ACM Press/Addison-Wesley Publishing Co., 1995.

[36] I.E. Sutherland. Sketch pad a man-machine graphical communication system. In *Proceedings of the SHARE design automation workshop*, pages 6–329. ACM, 1964.

[37] D.S. Tan, G.G. Robertson, and M. Czerwinski. Exploring 3d navigation: combining speed-coupled flying with orbiting. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 418–425. ACM, 2001.

[38] S. Ullah, N. Ouramdane, S. Otmane, P. Richard, F. Davesne, and M. Mallem. Augmenting 3d interactions with haptic guide in a large scale virtual environment. In *Proceedings of The 7th ACM SIGGRAPH International Conference on Virtual-Reality Continuum and Its Applications in Industry*, page 22. ACM, 2008.

[39] D. Valkov, F. Steinicke, G. Bruder, and K. Hinrichs. A multi-touch enabled human-transporter metaphor for virtual 3d traveling. In *3D User Interfaces (3DUI), 2010 IEEE Symposium on*, pages 79–82. IEEE, 2010.

[40] Colin Ware and Steven Osborne. Exploration and virtual camera control in virtual three dimensional environments. In *Proceedings of the 1990 symposium on Interactive 3D graphics*, I3D '90, pages 175–183, New York, NY, USA, 1990. ACM.