UNIVERSITÀ DEGLI STUDI DI PARMA

FACOLTÀ DI INGEGNERIA
CORSO DI LAUREA IN INGEGNERIA INFORMATICA

ANIMAZIONE DI GRUPPI DI AGENTI AUTONOMI BASATA SU COMPORTAMENTI PLAUSIBILI DI NAVIGAZIONE

AUTONOMOUS AGENT GROUP ANIMATION BASED ON PLAUSIBLE NAVIGATION BEHAVIORS

Relatore
Chiar.mo Prof. STEFANO CASELLI

Correlatori
Ing. JACOPO ALEOTTI
Ing. DARIO LODI RIZZINI

Tesi di Laurea di
VANESSA MANNI

23 Marzo 2011

Ai miei genitori,
Gerardo Manni e Maria Luisa Di Cesare

Nel momento della gioia per il raggiungimento di questo importante traguardo accademico sento il desiderio di ringraziare il mio Relatore, il Prof. Stefano Caselli che è stato punto di riferimento fin dal mio trasferimento all'università di Parma. Inoltre voglio ringraziare anche i miei correlatori, l'Ing. Jacopo Aleotti e l'Ing. Dario Lodi Rizzini, che mi hanno guidato e consigliato nel lavoro di tesi.

Vorrei ringraziare profondamente Francesco che mi è stato vicino fin dai primi giorni dei precorsi, quando ancora il cammino universitario era lungo e incerto; lo ringrazio perché mi è stato sempre accanto in ogni momento, anche quelli più duri, superando insieme a me le difficoltà. Lo ringrazio per aver condiviso e superato con me ansie e paure, ma soprattutto le soddisfazioni che questo primo traguardo accademico ci ha regalato. Ringrazio anche i suoi genitori e tutti i suoi parenti che sono per me come una seconda famiglia.

Ringrazio immensamente i miei genitori, entrambi assolutamente essenziali per la mia serenità, perché sono stati la mia guida, la mia ispirazione, la mia motivazione, la mia forza, anche in momenti disperati, e mi hanno sempre fatto sentire la loro presenza, nonostante la distanza. Li ringrazio per i consigli, che vorrei non mancassero mai, per le parole di conforto e anche per le chiacchiere fatte per farmi distrarre. Inoltre ringrazio Jessica, mia sorella, senza la quale non sarei la persona che sono. La ringrazio perché ha sempre cercato di farmi sentire il suo affetto, nonostante la distanza e ha sempre avuto una parola di conforto quando ne avevo bisogno.

Ringrazio tantissimo tutti i miei parenti, in particolare i miei nonni, dei quali sento costantemente la mancanza e la protezione. Ringrazio anche tutti gli zii e i cugini che, nonostante il poco tempo e le poche occasioni di vederci, sono sempre nel mio cuore.

Un profondo ringraziamento va anche a tutti gli amici con cui ho iniziato il percorso e l'avventura universitaria, in particolare Sara, Luca, Marco, Stefano, Andrea, Marta e Maurizio. Nonostante l'attuale lontananza hanno condiviso con me uno dei periodi più belli della mia vita.

Ultimo, ma non meno importante, è il ringraziamento per Andrea, Aurora, Emy, Andrea ed Enrico che in quest'ultimo periodo sono riusciti a farmi passare delle ore di svago e allegria.

Un professore di filosofia è in piedi davanti alla sua classe prima della lezione ed ha davanti a se alcuni oggetti. Quando la lezione comincia, senza proferire parola il professore prende un grosso vaso di vetro vuoto e lo riempie con delle pietre di 5/6 cm di diametro. Quindi chiede agli studenti se il vaso è pieno, ed essi annuiscono. Allora il professore prende una scatola di sassolini, e li versa nel vaso di vetro, scuotendolo appena. I sassolini, ovviamente, rotolano negli spazi vuoti fra le pietre. Il professore quindi chiede ancora se il vaso ora sia pieno, ed essi sono d'accordo. Gli studenti cominciano a sorridere, quando il professore prende una scatola di sabbia e la versa nel vaso. La sabbia riempie ogni spazio vuoto. "Ora", dice il professore, "voglio che voi riconosciate che questo è come la vostra vita. Le pietre sono le cose importanti - la famiglia, il partner, la salute, i figli - anche se ogni altra cosa dovesse mancare, e solo queste rimanere, la vostra vita sarebbe comunque piena. I sassolini sono le altre cose che contano, come il lavoro, la casa, l'auto. La sabbia rappresenta qualsiasi altra cosa, le piccole cose. Se voi riempite il vaso prima con la sabbia, non ci sarà più spazio per rocce e sassolini. Lo stesso è per la vostra vita; se voi spendete tutto il vostro tempo ed energie per le piccole cose, non avrete mai spazio per le cose veramente importanti. Stabilite le vostre priorità, il resto è solo sabbia!".

(Anonimo)

INDICE

Capitolo 1
Introduzione

1.1 Generalità e obiettivi della tesi

In seguito alla larga diffusione delle tecnologie e all'abbassamento dei prezzi degli elaboratori si è verificata negli ultimi decenni una grandissima espansione tecnologica. È, pertanto, di interesse generale capire, progettare, realizzare, sperimentare ed utilizzare sistemi artificiali di ogni genere (sia simulatori software che nuove realizzazioni hardware).

Con la presente tesi ci si è posti l'obiettivo di realizzare un'applicazione software con la quale rappresentare, utilizzare e modificare l'animazione di gruppi di agenti autonomi basata su comportamenti plausibili di navigazione. Nella pianificazione di sistemi multi agente il sistema deve essere in grado di esaminare un largo numero di evoluzioni del comportamento di ciascun agente. A questo proposito si è approfondito il movimento di agenti autonomi pilotati da comportamenti di guida semplici o combinati.

La simulazione del movimento di agenti può essere utilizzata in diversi ambiti:

- In robotica, ad esempio nel comportamento che definisce il movimento e le interazioni tra un gruppo di robot come il famoso AIBO (il robot della SONY presente in figura 1.1) dotato di un sistema di intelligenza artificiale capace di riconoscere oggetti, persone e di muoversi

autonomamente nello spazio. I comportamenti di tale robot sono analoghi a quelli studiati nel software che gestisce i movimenti degli agenti autonomi nel presente lavoro di tesi. Questo robot è stato utilizzato anche in competizioni come il RoboCup (campionato mondiale di calcio che si disputa fra robot autonomi) come mostrato in figura 1.2.

Figura 1.1: Un robot AIBO.

Figura 1.2: Una partita di calcio tra robot nella competizione Robocup.

- Nella progettazione di infrastrutture ed ambienti urbani è molto importante l'aspetto legato alla sicurezza nel movimento di folle in circostanze eccezionali; in primis il comportamento di masse di individui in caso di

eventi catastrofici in modo da poter pianificare al meglio gli spazi di fuga negli edifici pubblici come metropolitane, aeroporti, stadi, etc….

- In ambito bellico, nelle rivolte, nelle manifestazioni popolari o nelle azioni di pattugliamento si possono studiare strategie e conseguenze di un'azione militare in presenza di un numero elevato di uomini.
- Nei videogiochi, in particolare nel ruolo dei personaggi non giocanti che si muovono, agiscono autonomamente o reagiscono ad eventi generati dal giocatore. Al fine di assegnare una logica di comportamento al personaggio di un videogioco le possibilità sono molteplici ma, la maggior parte delle volte, vengono utilizzate delle porzioni di codice all'interno delle quali sono definite le "istruzioni" da eseguirsi al verificarsi di determinati "eventi".

L'obiettivo principale della tesi è quello di realizzare agenti che riescano a muoversi in modo autonomo all'interno di una scena. Il sistema dovrà prevedere agenti capaci di individuare ostacoli posti nella scena ed evitarli. Inoltre dovranno riuscire ad evitarsi reciprocamente, in modo da non collidere gli uni con gli altri, anche in presenza di tanti agenti in una zona ristretta. Gli agenti dovranno avere uno o più target da raggiungere (nel caso di più target questi dovranno avere un ordine in modo da creare delle semplici traiettorie). Tutte le richieste dovranno essere soddisfatte anche contemporaneamente secondo un principio di priorità o di somma ponderata. Un ulteriore obiettivo della tesi è riuscire ad integrare il sistema con un simulatore grafico 3D in modo da rendere il risultato finale visivamente gradevole.

1.2 Organizzazione della tesi

La tesi è organizzata in diversi capitoli, dei quali il primo è incaricato di definire le generalità e gli obiettivi.
Il secondo capitolo è utilizzato per approfondire il movimento degli agenti autonomi nei possibili comportamenti realizzabili. Viene approfondito il significato di sistema multi agente e le diverse categorie di agenti; viene inoltre

presentato un modello semplice per realizzare gli agenti nella scena basato su una gerarchia a tre livelli (selezione azione, sterzo e movimento).

Nel terzo capitolo sono illustrati gli strumenti software utilizzati, tra cui soprattutto le librerie open source Opensteer ideate da Craig Reynolds, che sono la base utilizzata in questa tesi per costruire comportamenti di guida interessanti. Con queste librerie è stato realizzato il motore dell'applicazione (presentato in una rappresentazione di base in 2D). In un secondo momento il sistema è stato integrato con un ambiente 3D esistente per il quale sono stati utilizzati Horde3D per il motore del simulatore, ColladaConverter per riuscire a convertire e integrare nell'ambiente oggetti esterni, Google SketchUp Pro per la realizzazione di elementi architettonici ed edifici, 3ds Max 9 per animare i modelli di persone umane, e infine CubeMapGen per creare lo sfondo in lontananza attraverso uno skybox.

Nel quarto capitolo viene presentato il lavoro svolto per realizzare il sistema. Vengono illustrati sia la parte relativa al motore che stabilisce il movimento degli agenti, sia la parte di integrazione col motore grafico, soffermandosi sulle difficoltà riscontrate e sull'analisi delle prestazioni.

Infine sono presentate le conclusioni del lavoro svolto e la bibliografia.

Capitolo 2
Comportamento di agenti autonomi

2.1 Il comportamento autonomo

Nella società attuale ci troviamo di fronte a sistemi di elaborazione sempre più complessi, interconnessi tra di loro, così da spingere la ricerca a realizzare sistemi software costituiti da entità autonome detti agenti; tali agenti hanno la capacità di interagire tra loro in ambienti complessi e possono essere impiegati in svariati ambiti, come il commercio elettronico, il grid computing, i web service, la robotica, i videogames e la simulazione.

Volendo fare un esempio pratico, consideriamo i videogiochi e le scene di animazione; in questi casi uno dei requisiti principali che i personaggi devono avere è la capacità di muoversi all'interno dell'ambiente circostante in maniera realistica e individuale. Molto spesso nella realizzazione di videogiochi, o nella creazione di realtà virtuali si ha la necessità di avere a disposizione agenti capaci di determinare autonomamente le proprie azioni. Tali agenti vengono anche chiamati "personaggi autonomi" (nell'ambito strettamente video-ludico sono anche detti personaggi "non-giocanti") perché possono prendere decisioni in modo indipendente e spesso in risposta a modifiche dell'ambiente circostante.

Ad esempio in un videogioco dove sia presente una guardia a sorvegliare un ingresso, si può notare come questa continuerà a muoversi nella zona circostante fino a quando un elemento estraneo non si avvicini nel suo raggio d'azione; in tal

caso potranno verificarsi diversi comportamenti della guardia come avvicinarsi all'intruso o attaccarlo. Un esempio del genere è realizzato in molti videogiochi, tra cui il noto Metal Gear Solid (1).

Figura 2.1: Immagine tratta dal videogioco Metal Gear Solid nel quale i personaggi non-giocanti reagiscono ad azioni esterne e variazioni dell'ambiente.

Nell'immagine in figura 2.1 si vedono due guardie che sorvegliano la zona mentre in basso il personaggio controllato dal giocatore è nascosto dietro al cespuglio, aspettando il momento giusto per eludere le guardie.
Questo è solo un esempio tra tanti. Il framework denominato XNA (2) permette di costruire giochi anche per piattaforme come Windows e Xbox 360 (3). I giochi realizzati con questo framework di basano su un modello semplice di intelligenza artificiale, nel quale gli agenti sono capaci di percepire e reagire all'ambiente circostante in modo a autonomo.
L'insieme dei comportamenti che il carattere non-giocante può avere lo differenzia dai personaggi di un film animato le cui azioni sono predeterminate. Un personaggio autonomo deve combinare gli aspetti di un robot autonomo con l'abilità di un attore umano in un teatro d'improvvisazione. Questi agenti non sono robot e non sono certamente degli attori umani, ma condividono alcune proprietà di entrambi.
Dato che il termine "agente autonomo" è usato in molti contesti, proviamo a individuare la terminologia che meglio si adatta al nostro contesto in relazione ad altri campi di studio.

Un agente autonomo può esistere in isolamento, o può trovarsi in condivisione con altre entità; inoltre può essere reattivo (istintivo, guidato da stimoli) o deliberativo ("intellettuale" nel senso classico di agente intelligente); infine può riguardare esclusivamente informazioni astratte ("softbot[1]", "knowbot[2]", "agente di informazioni") oppure può essere incorporato in una manifestazione fisica (un tipico robot industriale o autonomo). Combinazioni di *posizioni*, *reazioni*, e *rappresentazioni* definiscono diverse classi di agenti autonomi.

Nella categoria di *posizioni* gli agenti integrati sono associati solitamente ai robot autonomi, dispositivi meccanici esistenti nel mondo reale. A volte i robot sono studiati tramite simulazione computazionale, ma questa pratica è vista con diffidenza dai puristi nel campo della robotica, perché la simulazione può divergere dalla realtà in modo imprevedibile.

I personaggi autonomi trattati in questo lavoro sono: posizionati, rappresentati, reattivi e agenti virtuali. In questo caso il termine virtuale, come realtà virtuale (4), verrà utilizzato per indicare che questi agenti, invece di essere simulazioni di un dispositivo meccanico nel mondo reale, sono invece veri agenti in un mondo virtuale.

Occorre infine chiarire il significato attribuito nel nostro studio al termine *comportamento*: in generale può esprimere la complessa azione di un uomo o di un animale basato sulla volontà oppure sull'istinto, può indicare la prevedibile azione di un sistema meccanico semplice o la complessa azione di un sistema caotico; spesso, nella realtà virtuale e nelle applicazioni multimediali, il termine comportamento è impropriamente usato come sinonimo di "animazione". In questo lavoro il termine comportamento è invece usato per riferirsi alle azioni di improvvisazione autonome da parte degli agenti. Saranno fornite agli agenti delle combinazioni di comportamenti di guida che possono essere utilizzate per raggiungere obiettivi di livello superiore (ad esempio: arrivare da un punto a un

[1] "softbot", un nuovo robot che sarà in grado di cambiare forma e di muoversi all'interno del corpo umano. Questi robot si muoveranno in modo diverso da qualsiasi altro robot esistente, per la precisione si muoveranno come un bruco.

[2] Knowbot è l'acronimo di Knowledge-Based Object Technology. I webbot, conosciuti anche come robot del web, sono applicazioni software che eseguono task automatici su Internet. In genere i bot eseguono attività che sono strutturalmente semplici e ripetitive, ad una velocità molto più elevato di quanto sarebbe possibile per un uomo. Knowbot è un tipo di bot che raccoglie automaticamente determinate informazioni dai siti web.

altro, evitando gli ostacoli, seguendo un determinato percorso, aggregandosi a un gruppo di personaggi...). Lo scopo di questa tesi è presentare tecniche generali nell'ambito dei videogiochi e della simulazione di folle per la progettazione di sistemi multi agente.
In particolare saranno presentati metodi per la progettazione di agenti singoli in grado di prendere decisioni razionali in condizioni di incertezza, e per la progettazione di sistemi composti da più agenti. Sarà inoltre presentato uno strumento per lo sviluppo e la realizzazione di un sistema basato su più agenti.

2.2 Sistemi multi agente

I sistemi multi agente, anche noti come sistemi ad agenti multipli, sono un insieme di entità autonome situate in un certo ambiente in modo da poter interagire tra di loro attraverso una opportuna organizzazione. I sistemi ad agenti multipli costituiscono un'interessante tipologia di modellazione di società ed hanno a questo riguardo vasti campi d'applicazione, che si estendono fino alle scienze umane e sociali (come in economia, sociologia, etc.).
La struttura di un sistema multiagente può essere delineata in alcuni punti fondamentali: essenziale è dotare le nostre entità di capacità di pianificazione e decisione, in modo da poter interagire correttamente con l'ambiente che li circonda; bisogna poi fornire un modello cognitivo, ovvero l'insieme delle conoscenze dell'agente (sapere quali possono essere gli ostacoli, o i target da raggiungere), delle sue percezioni (riuscire a vedere un ostacolo, o altri agenti e riconoscerli come tali) e infine avere un insieme di obbiettivi. Tutto questo costituisce l'insieme di intenzioni che comporteranno le successive azioni.
L'origine di questi sistemi è da attribuirsi all'evoluzione di diverse discipline legate all'informatica, tra queste gli aspetti decisionali dell'agente nell'ambito dell'intelligenza artificiale (5), l'evoluzione verso un sistema sempre più autonomo per l'ingegneria del software e l'interazione tra agenti nei sistemi distribuiti.
Nel seguito esamineremo gli aspetti che affiancano i sistemi multi agente all'intelligenza artificiale per descrivere i caratteri che hanno portato all'evoluzione di questa disciplina. Tale studio ci permetterà in un secondo

momento di comprendere al meglio il nostro strumento di sviluppo basato su sistemi multi agente.

2.2.1 Diverse categorie di agenti

È possibile distinguere tra svariate categorie di agenti secondo diversi criteri: agenti cognitivi, reattivi e a comportamento riflesso.

Tale distinzione è stata fatta a seguito della contrapposizione di due scuole di pensiero relative ai sistemi multiagente (6): la prima, partendo da una prospettiva più sociologica, sostiene un approccio basato su insiemi di agenti "intelligenti" collaborativi; la seconda studia un comportamento "intelligente" di un insieme d'agenti non intelligenti.

La differenza tra cognitivo e reattivo dipende dal tipo di percezione del mondo di cui gli agenti dispongono. Si può avere infatti una rappresentazione simbolica o subsimbolica. Nel primo caso, se il singolo agente è in grado di formulare ragionamenti a partire dalla rappresentazione simbolica del mondo, si parlerà di agente cognitivo; se invece si dispone d'una rappresentazione limitata alle percezioni, si parlerà di agente reattivo. In quest'ultimo caso l'agente non possiede a priori la conoscenza di tutto il mondo circostante ma la apprende esplorandolo ed interagendo con esso. Quest'ultimo è il tipo di agente che verrà utilizzato nella parte progettuale della tesi.

Per quanto riguarda i comportamenti intenzionali (perseguire scopi espliciti) e quelli legati a percezioni, gli agenti possono assumere decisioni derivanti esplicitamente all'interno o provenienti dall'ambiente esterno.

Gli agenti cognitivi sono il più delle volte intenzionali, cioè hanno scopi prefissati che tentano di realizzare. Si possono pure trovare talvolta agenti detti modulari i quali, anche se possiedono una rappresentazione del proprio universo, non hanno tuttavia degli scopi precisi.

Gli agenti reattivi possono a loro volta essere suddivisi in agenti impulsivi e tropici. Un agente impulsivo avrà una missione ben determinata e, qualora percepisca che l'ambiente non risponde più allo scopo che gli è stato affidato, agirà di conseguenza. L'agente tropico, dal canto suo, reagisce solo allo stato

locale dell'ambiente. La fonte della motivazione è nel primo caso interna (agenti impulsivi che hanno una "missione"), nel secondo caso è legata esclusivamente all'ambiente. Volendo fare un esempio relativo alla simulazione di folle gli agenti tropici possono essere rappresentati dalla gente che scappa in presenza di un incendio, mentre l'agente impulsivo può essere rappresentato come il vigile del fuoco che ha il compito di spegnere le fiamme, qualora si verifichi un incendio.

2.3 Alcune problematiche sui sistemi multiagente

La creazione dei sistemi multi agente evidenzia diverse problematiche, tra cui in particolare la rappresentazione dell'ambiente, la pianificazione delle attività svolte dagli agenti e la collaborazione degli stessi.
Decisamente importante è la relazione dell'agente rispetto al mondo esterno, cioè la rappresentazione del modello cognitivo dell'agente. L'aspetto fondamentale è la capacità di prendere decisioni collegate alle percezioni dell'agente, anche in relazione agli altri agenti presenti. Appare fondamentale, in una società costituita da molteplici agenti, che ogni individuo riesca a raggiungere i propri obbiettivi.
Un'altra problematica rilevante è quella dell'adattamento che può essere inteso in senso individuale (costituendo un vero e proprio apprendimento dell'agente) o in senso collettivo ovvero come una sorta di evoluzione.
Infine rimane la questione della realizzazione effettiva e dell'implementazione che può essere fatta attraverso la strutturazione di linguaggi di programmazione.

2.4 Comportamenti direttivi

Nel sistema sviluppato in questo lavoro l'interazione tra gli agenti e tutto ciò che li circonda è gestito attraverso comportamenti direttivi che guidano le azioni degli agenti nella scena.
I comportamenti direttivi attingono ad una lunga storia di ricerche correlate in diversi campi: macchine autonome, servomeccanismi, e teoria del controllo che hanno le loro radici nella cibernetica sviluppatasi intorno al 1940 (7). Il termine cibernetica proviene dal greco e significa timoniere. Nel 1980 Valentino Braitenberg condusse esperimenti su una serie di veicoli con comportamenti

sempre più complessi. David Zeltzer nello stesso periodo iniziò ad applicare tecniche e modelli di intelligenza artificiale nell'animazione. Nel 1987, fu creato da Reynolds (8) un modello comportamentale animato di stormi di uccelli (noto anche come boids).

Le ricerche furono portate avanti in tre categorie: robotica, intelligenza artificiale e vita artificiale, anche se in alcuni casi la distinzione è sottile.

Nel 1987 il modello di agenti che rappresentavano greggi, mandrie, scolaresche e movimenti di gruppo connessi, articolato con un comportamento complesso, fu decomposto in tre comportamenti di guida semplici a livello individuale.

Nel 1994 G. Keith Still (9) ha modellato grandi folle umane usando per ciascun individuo uno schema comportamentale di guida. Nello stesso periodo Karl Sims (10), utilizzando un algoritmo genetico modificato, sviluppò il cervello e il corpo di creature artificiali per i vari tipi di movimento e per l'obiettivo di ricerca.

James Cremer (11) nel 1996 insieme a dei suoi colleghi creò i driver autonomi per gestire il traffico automobilistico all'interno di un ambiente interattivo con simulatori di guida. Dave Pottinger (12) nel 1999 fornì una descrizione dettagliata di direzione e coordinamento nei gruppi di personaggi nei giochi.

Queste sono state le basi che hanno permesso la realizzazione delle librerie Opensteer che consentono di far interagire agenti indipendenti per mezzo di comportamenti direttivi (anche detti comportamenti di guida).

2.5 Gerarchia a tre livelli

Il comportamento di tipo autonomo può essere meglio compreso dividendolo in più livelli: il più basso di locomozione, il livello medio di comportamenti di guida o sterzo, e il più alto di definizione degli obiettivi e strategia. Questi livelli sono pensati per chiarire e specificare il meccanismo complesso che guida le decisioni degli agenti autonomi. La figura 2.2 mostra l'organizzazione logica del comportamento degli agenti per il movimento in una gerarchia di tre strati: *selezione azione, sterzo,* e *movimento.*

Si noti che mentre la gerarchia comportamentale qui presentata è destinata ad essere ampiamente applicabile a comportamenti di movimento, non è adatta per

altri tipi di azioni autonome, per esempio i comportamenti di conversazione di un "chatterbot[3]" richiedono una struttura decisamente diversa.

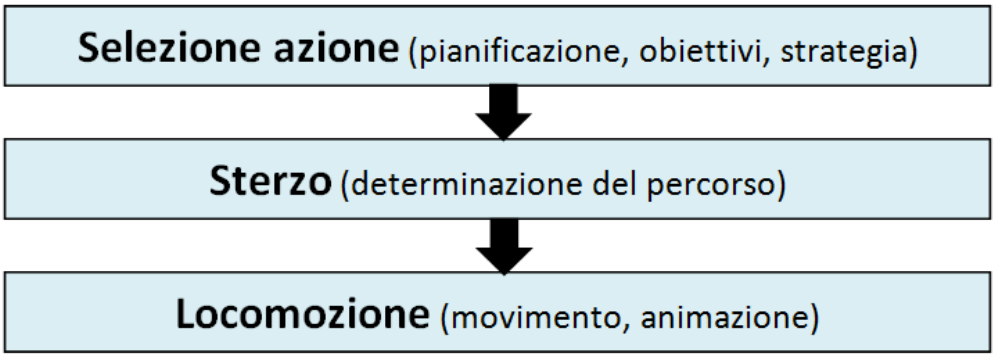

Figura 2.2: Una gerarchia di comportamenti movimento

Si considerino, ad esempio, alcuni cowboy che tengono una mandria al pascolo su un campo; una mucca si allontana dal branco; il cowboy leader dice ad un cowboy di andare a recuperare la mucca; il cowboy comanda al suo cavallo di seguire la mucca, possibilmente evitando gli ostacoli lungo la strada. In questo esempio, il cowboy leader rappresenta la *selezione azione*: lo stato della situazione è cambiato (una mucca ha lasciato la mandria) ed è stato fissato un obiettivo (recuperare la mucca allontanata). Il livello di governo è rappresentato dal cowboy, che scompone l'obiettivo in una serie di sotto-obiettivi semplici (avvicinarsi alla mucca, evitare gli ostacoli, recuperare la mucca). Un obiettivo secondario corrisponde al comportamento direttivo della squadra formata da cavallo e cowboy. Utilizzando diversi segnali di controllo (comandi vocali, speroni, briglie), il cowboy dirige il cavallo verso il bersaglio. In termini generali, questi comandi corrispondono a segnali ben precisi come: andare più veloce, più lento, girare a destra, svoltare a sinistra, e così via. Il cavallo implementa il livello di *locomozione*.

Assumendo i comandi di controllo del cowboy come segnali di input, il cavallo si muove nella direzione indicata. Questo movimento è il risultato della complessa interazione tra la percezione visiva del cavallo, il suo senso di equilibrio, e

[3] Un chatbot (o chatterbot) è un programma per computer progettato per simulare una conversazione con uno o più utenti umani attraverso modalità testuale o audio.

l'applicazione di coppie attraverso i muscoli alle articolazioni dello scheletro. Da un punto di vista ingegneristico, il movimento su gambe (o zampe) è un problema molto complesso, ma né il cowboy né il cavallo hanno bisogno di prestargli particolare attenzione.
Per capire il funzionamento degli agenti autonomi ci concentreremo sullo *sterzo*, ovvero lo strato intermedio della gerarchia comportamentale.
Verrà descritto un semplice modello dello strato di locomozione solo ad un livello sufficiente a fornire una base concreta per la discussione di comportamenti di guida diversi. Nel seguito faremo qualche cenno alla *selezione azione*, soprattutto in relazione alla combinazione di comportamenti di guida di base.
I comportamenti di guida che sono stati sviluppati in questo progetto sono relativi al movimento veloce degli agenti. Questo va precisato al fine di chiarire che gli agenti che osserveremo avranno una velocità relativamente elevata; inoltre i comportamenti di guida dovranno prevedere il futuro e considerare il sopravvenire di situazioni nuove e le loro conseguenze. Ciò non si discosta molto dalla realtà, in quanto quando si è alla guida di una macchina e si tenta di fare un sorpasso, osservando l'ambiente, si deve prevedere la possibilità di incontrare un veicolo che percorre la strada in senso opposto.
Quello che ipotizzeremo per i nostri agenti non sarà una conoscenza del futuro, ma solo la capacità di prevedere una sua possibile evoluzione ed agire di conseguenza.

2.5.1 Lo strato di locomozione

Lo strato di locomozione è alla base della gerarchia a tre livelli che è stato appena descritto. Questo strato traduce i segnali provenienti dal livello di sterzo in movimenti dell'agente. Questo movimento è soggetto ai vincoli imposti dal modello fisico applicato al corpo del personaggio. Come visto nell'esempio del cowboy, il cavallo rappresenta il livello di locomozione. Le decisioni del cowboy sono trasmesse al cavallo per mezzo di segnali di controllo semplici che questo è in grado di interpretare e tradurre in movimento.
È importante distinguere tra livello di sterzo e di locomozione per comprendere che si può cambiare il livello di locomozione lasciando il resto inalterato. Ad

esempio se immaginiamo di prendere il cowboy e metterlo su una motocicletta (al posto del cavallo), vediamo che la selezione obbiettivo e il comportamento di guida rimangono invariati. L'unica cosa che cambia è il meccanismo per la mappatura dei segnali di controllo (andare più veloce, girare a destra, ...) che ora devono essere adeguati alla moto. Il ruolo del pilota resta invariato, abbiamo solo sostituito il sistema di locomozione (zampe, ossa, muscoli) con un altro differente (motore, ruote, freni).
Questa osservazione, che può sembrare banale, serve a sottolineare la possibilità di creare un comportamento di guida indipendente dal sistema di locomozione adottato, a patto che si stabiliscano determinate convenzioni idonee per i segnali di controllo.
È possibile infatti concepire e progettare agenti autonomi che basano il loro movimento su alcune leggi fisiche in modo tale da rendere più realistica la loro animazione e il comportamento; in alternativa è possibile avvalersi di modelli più semplici:

1. utilizzare figure pre-animate che eseguono ciclicamente delle animazioni di camminata
2. considerare un ulteriore livello di astrazione, e rappresentare gli agenti come semplici cerchi che si muovono nell'ambiente.

Questi livelli di astrazione non sono molto diversi da quelli che vengono usati in fisica per studiare il moto dei corpi, ed è proprio di questi due modelli più semplici che ci occuperemo in questa tesi.

2.6 Un modello di veicolo semplice

In questo lavoro di tesi il concetto di base consiste nel considerare i comportamenti di guida indipendentemente dal tipo di locomozione adottata. Sarà pertanto presentato un modello di locomozione molto semplice, in modo da poterci concentrare sui comportamenti di sterzo. Il modello di locomozione sarà quindi raffigurato da un veicolo astratto, che può rappresentare una vasta gamma di possibilità concrete: dai mezzi di trasporto (automobili, aerei, navi spaziali…)

alle rappresentazioni di gruppi (folle, mandrie…). La decisione di scegliere una rappresentazione così semplicistica del veicolo permetterà di generalizzare il lavoro a tutte queste possibilità.

Questo veicolo è basato sull'approssimazione di massa puntiforme e, sebbene non sia un modello fisico particolarmente interessante (in quanto le masse puntiformi nel mondo reale non esistono), è un modello computazionalmente economico perchè una massa puntiforme ha velocità ma non ha momento d'inerzia.

Un punto materiale è definito da una posizione e da una massa, inoltre il nostro modello sarà dotato di velocità che verrà modificata da forze in genere auto applicate e quindi limitate. Ad esempio una tipica forza che regola la velocità di un veicolo come una bicicletta è la spinta, generata dai muscoli del ciclista e per questo limitata in grandezza proprio dalle capacità stesse del ciclista.

Per il modello di veicolo semplice che utilizzeremo questo concetto sarà riassunto da un unico parametro che rappresenterà la massima forza applicabile (max_force). La maggior parte dei veicoli sono caratterizzati da una velocità di punta, e in genere questa limitazione è dovuta all'interazione tra l'accelerazione generata dalla spinta e la decelerazione dell'attrito e della resistenza viscosa.

Per la simulazione realistica di tutte queste forze, il modello di veicolo semplice include un parametro di velocità massima (max_speed). Questo limite di velocità viene imposto come troncamento del vettore velocità del veicolo. Infine il modello di veicolo semplice include un parametro di orientamento che insieme alla posizione permette di localizzare e far muovere il modello geometrico in modo adeguato. Ad esempio con queste informazioni si potrà pensare di aggiungere in un secondo momento un'animazione più complessa sfruttando i dati di orientamento e posizione del veicolo semplice.

Per un modello di veicolo 3D i valori della posizione e della velocità sono vettori con tre componenti, mentre per il valore dell'orientamento si avrà un insieme di tre vettori (o una matrice 3x3). Per un veicolo 2D i vettori di posizione e velocità hanno ciascuno due componenti mentre il valore dell'orientamento sarà espresso con due vettori in due dimensioni, oppure può essere rappresentato con uno scalare in riferimento all'angolo.

Ad ogni passo della simulazione comportamentale vengono determinate le forze di governo (con i limiti max_force) che vengono applicate al veicolo di massa

puntiforme. Questo produce un'accelerazione pari alla forza di guida diviso la massa del veicolo; tale valore viene sommato alla velocità già esistente per acquisire il nuovo valore di velocità da attribuire al veicolo (da notare che il valore ottenuto sarà sempre troncato al valore max_speed).
Il modello di veicolo semplice in questo modo riesce a mantenere una velocità locale incrementale rispetto ai passi temporali precedenti. Il sistema di coordinate è definito in termini di quattro vettori: un vettore di posizione e tre vettori di direzione dell'agente, come vettori di base dello spazio.
Il veicolo si potrà muovere in tre direzioni: avanti o indietro sull'asse X, lateralmente sull'asse Z, verso l'alto o il basso sull'asse Y. Con i vettori di base, si ottiene uno spazio vettoriale R^3 come mostrato in figura 2.3.

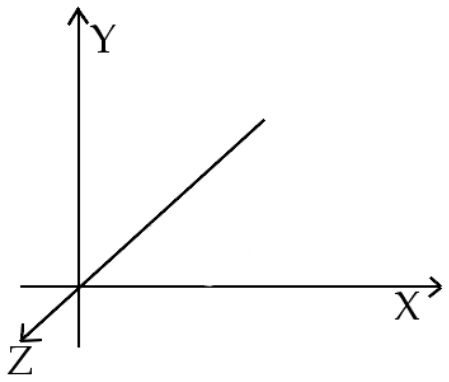

Figura 2.3: Rappresentazione di uno spazio vettoriale R^3.

Per far in modo che il veicolo scivoli sulla superficie, vogliamo vincolare la posizione del veicolo al piano ed allinearlo su un asse normale. In più vogliamo che la velocità sia puramente tangenziale alla superficie.
In questo modello di veicolo semplice il segnale di controllo che passa dai comportamenti di guida al livello di locomozione consiste esattamente in una grandezza vettoriale: la forza di governo desiderata. Per modelli di veicoli realistici il set dei segnali di controllo sarebbe molto diverso. Per esempio un automobile ha un volante, un acceleratore e un freno, ciascuno dei quali può essere rappresentato da quantità scalari.
È possibile associare un generico vettore di forza di governo a questi segnali scalari: la componente laterale del vettore di guida può essere associata al segnale

relativo allo sterzo dell'auto, la componente in avanti del vettore di direzione può essere interpretata come segnale da associare all'acceleratore (se il valore è positivo), o al freno (per valore negativo). L'associazione è asimmetrica in quanto l'azione motrice e frenante producono effetti non proporzionali alle intensità applicate, come mostrato in figura 2.4.

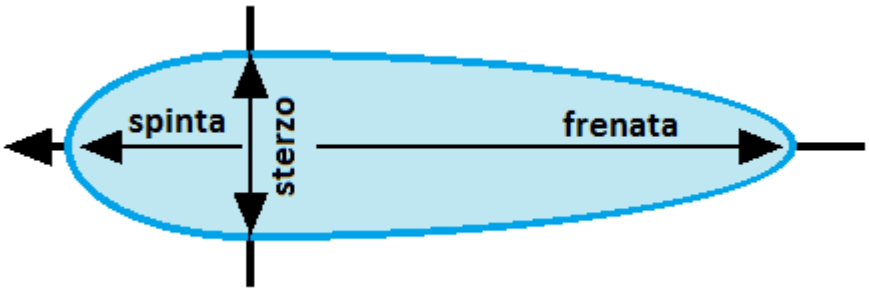

Figura 2.4: Forze di sterzo asimmetriche

Questo modello permette di girare anche quando la sua velocità è pari a zero, mentre la maggior parte dei veicoli reali non può farlo. Tale problema può essere risolto ponendo un ulteriore vincolo sul cambiamento di orientamento o limitando la componente laterale dello sterzo a bassa velocità, o simulando il momento d'inerzia.

2.7 Comportamenti di guida

In questo paragrafo vengono illustrati gli specifici comportamenti di guida supponendo di utilizzare il modello di locomozione "veicolo semplice" precedentemente definito, con la parametrizzazione in un unico vettore di forza. Il comportamento dello sterzo è descritto calcolando geometricamente un vettore che rappresenta la forza di governo desiderata ed anche se la grandezza di questi vettori dovesse essere elevata, il valore sarà comunque troncato al max_force del modello del veicolo.

2.7.1 Seek

Il comportamento **seek** (figura 2.5), ovvero la ricerca da parte dell'agente di un obiettivo di tipo statico, indirizza l'agente verso una determinata posizione nello spazio. Questo comportamento guida il veicolo in modo tale da allineare la sua velocità in modo radiale rispetto al bersaglio. Da notare che questa situazione è diversa da quella che produrrebbe una forza attrattiva (come la gravità) che avrebbe un percorso orbitale attorno al punto di destinazione. La velocità desiderata è un vettore nella direzione dal veicolo verso il bersaglio, la cui lunghezza potrebbe essere max_speed oppure potrebbe essere l'attuale velocità del personaggio, a seconda della particolare applicazione. Il vettore di sterzo è la differenza tra la velocità desiderata e quella attuale del personaggio. Se un personaggio continua a cercare il bersaglio anche dopo averlo raggiunto, finirà per passargli attraverso e poi tornare indietro per avvicinarsi nuovamente. Questo movimento ricorda quello di una falena che ronza intorno a una lampadina.

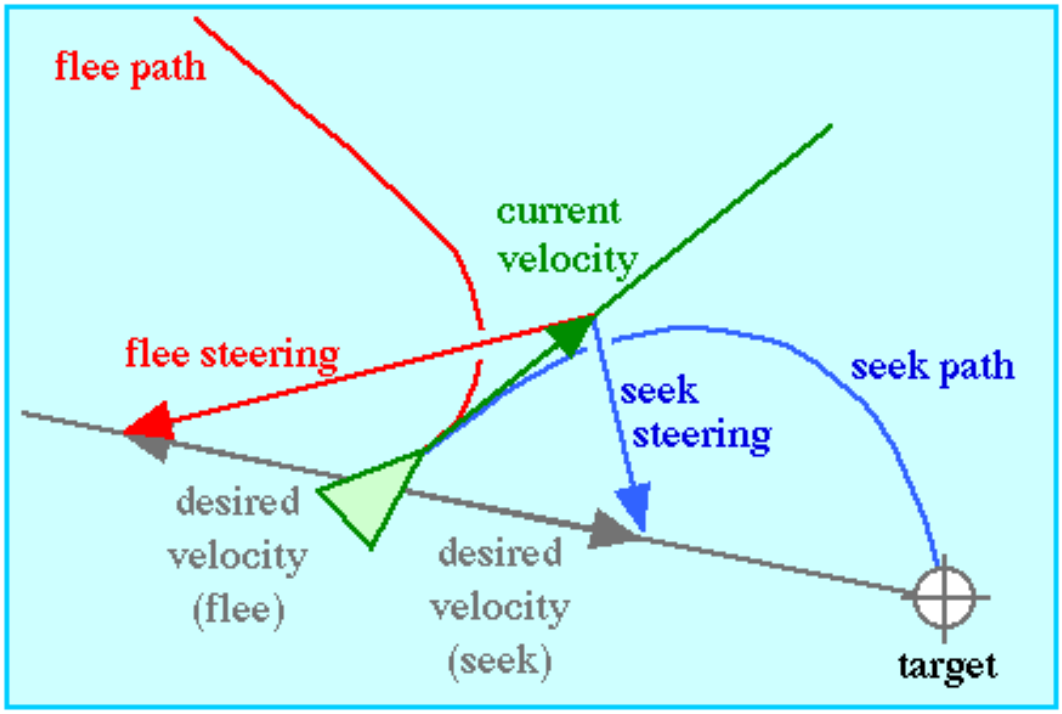

Figura 2.5: **Cercare** e **fuggire**

2.7.2 Flee

Il comportamento **flee** (ovvero fuggi) è semplicemente l'inverso della ricerca e indirizza il personaggio in modo che la sua velocità sia allineata radialmente in fuga dal bersaglio.

2.7.3 Pursuit

Pursuit (o inseguimento) è simile alla ricerca con la differenza che il target è un altro personaggio in movimento. Per un utilizzo efficace di questo comportamento è necessaria la previsione della futura posizione dell'obbiettivo. A tal fine si utilizza un meccanismo semplice che viene rivalutato ad ogni passo della simulazione. Ad esempio se si basa sulla velocità lineare corrisponde al presupposto che l'obiettivo da raggiungere non possa girare durante l'intervallo di previsione. Questa ipotesi è quasi sempre scorretta in quanto sarà verificata solo per un tempo estremamente ridotto. Per ottenere la previsione della posizione di un veicolo in anticipo di T unità di tempo si può scalare la sua velocità di un valore pari a T e compensare la sua posizione attuale. A questo punto il comportamento di inseguimento è semplice da realizzare applicando la ricerca alla posizione prevista per il target. L'aspetto chiave è proprio il metodo utilizzato per valutare la previsione della posizione futura. Idealmente T sarebbe il tempo dall'intercettazione, ma tale valore è sconosciuto in quanto il target può fare manovre imprevedibili e arbitrarie. Si può allora considerare il valore di T come una costante che, seppur non ottimale, produrrebbe un risultato migliore rispetto a quello ottenuto utilizzando semplicemente il comportamento di ricerca a bersaglio fisso (quindi T=0). Per ottenere prestazioni ragionevoli T dovrebbe essere maggiore quando l'inseguitore è lontano dall'obiettivo, mentre può essere minore quando si trova nelle vicinanze. Per considerare una stima semplice della qualità possiamo valutare T=D*c dove consideriamo D la distanza tra inseguitore e preda, mentre c è un parametro per tener conto della rotazione. Per una stima più complessa si dovrebbe tener conto delle posizioni reciproche tra inseguitore e preda (se l'inseguitore è davanti, dietro o di lato alla preda). Questi parametri possono essere espressi in termini di semplice prodotto. In questo caso bisogna

ricordarsi di ridurre T (ad esempio a zero) quando l'inseguitore si allinea di fronte alla sua preda.

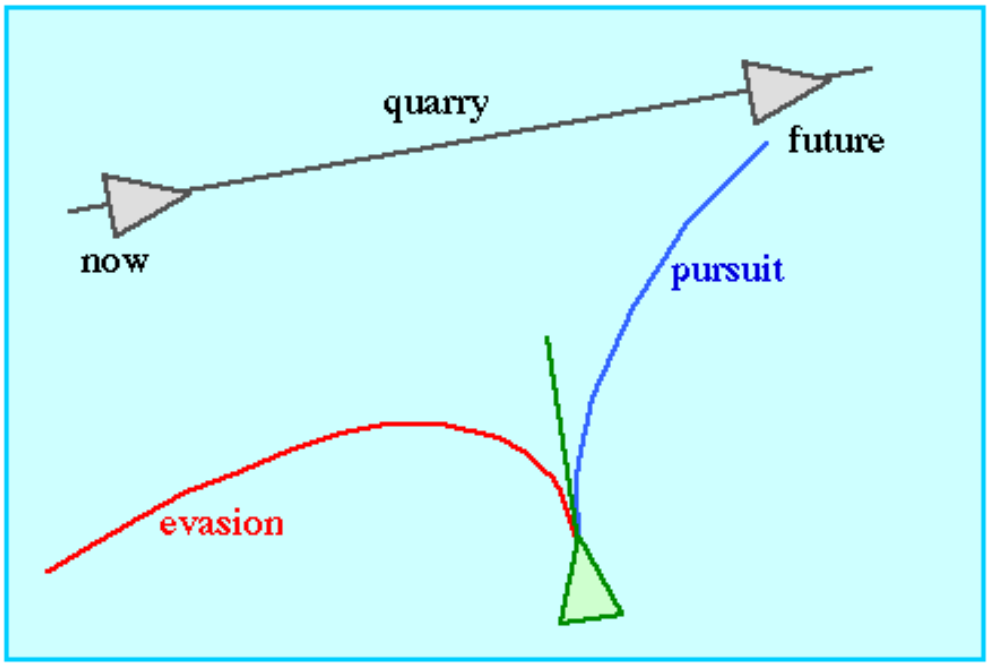

Figura 2.6: **Inseguimento** ed **Evasione**

2.7.4 Evasion

Il comportamento **evasion** è analogo a quello di inseguimento, ma in senso opposto, infatti viene considerata la posizione futura dell'obbiettivo per allontanarsi da esso. Le versioni di evasione adottate sono poco complesse ma non ottimali. Nei sistemi naturali l'evasione è spesso intenzionalmente non ottimale al fine di essere imprevedibile, consentendo di sventare le strategie di ricerca predittiva.

2.7.5 Arrival

Il comportamento **arrival** è identico al comportamento di ricerca nella situazione di veicolo molto lontano dall'obbiettivo. Quello che cambia è il movimento quando ci si avvicina all'obbiettivo, infatti una volta raggiunto il target, invece di attraversarlo a tutta velocità rallenta fino a fermarsi coincidendo con il target

(come mostrato in figura 2.7). La distanza dalla quale inizia il rallentamento è un parametro del comportamento. L'implementazione è molto simile a quella del comportamenti di ricerca. La velocità desiderata è determinata dal tipo di puntamento verso il bersaglio, e se troppo elevata viene sempre troncata al valore max_speed. Un esempio di questo comportamento riferito al mondo reale potrebbe essere quello di un'automobile che si dirige verso un incrocio e si ferma proprio vicino al semaforo.

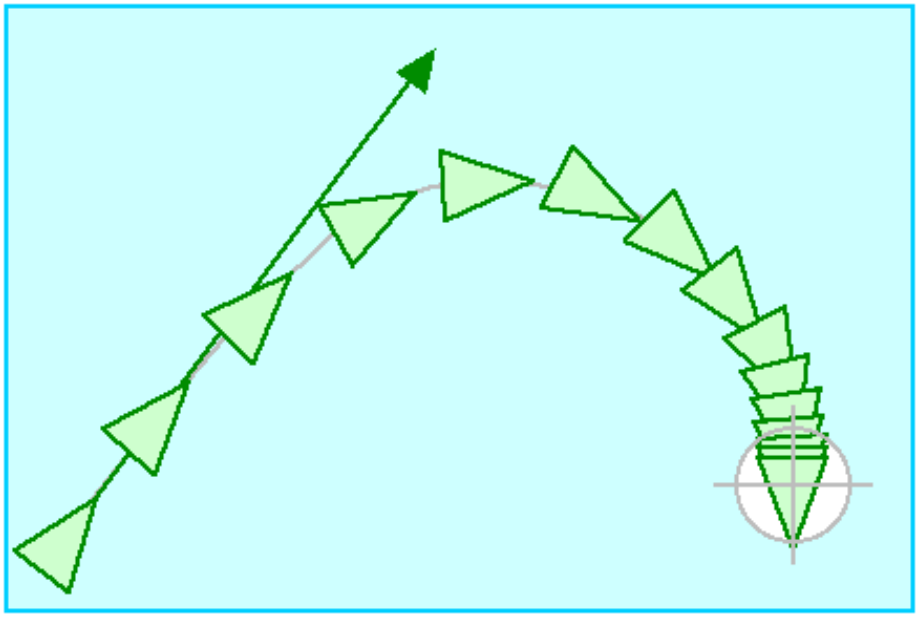

Figura 2.7: **Arrivo**

2.7.6 Obstacle avoidance

Il comportamento **obstacle avoidance** attribuisce al veicolo la capacità di muoversi in un ambiente pieno di ostacoli e riuscire a schivarli. Bisogna sottolineare le differenze tra il comportamento di evitare ostacoli e quello di fuggire. Quest'ultimo infatti provocherà l'allontanamento del veicolo, in modo da aumentare la distanze da una determinata posizione, mentre l'evitare ostacoli interviene solo quando l'ostacolo si trova proprio di fronte al personaggio che rischia di andarci a sbattere. Ad esempio se un automobile si trova a costeggiare parallelamente una parete per il comportamento di fuga si posizionerebbe in direzione opposta al muro e proseguirebbe in quella direzione, mentre nel caso di

evitare ostacoli, l'automobile non cambierebbe la sua direzione, in quanto il pericolo di collisione sarebbe inesistente. Per la realizzazione di tale comportamento si farà l'ipotesi semplificativa che sia il veicolo che l'ostacolo siano ragionevolmente approssimati a delle sfere, anche se il concetto di base può essere facilmente esteso a forme diverse. Questa ipotesi di geometria riguarda non solo l'evasione degli ostacoli, ma più in generale il rilevamento delle collisioni. Se pensiamo a un aereo che deve evitare una montagna, gli oggetti non sono di forme sferica, ma se immaginiamo che entrambi siano delimitati da due sfere, basterebbe che la sfera-aereo riesca ad evitare la sfera-montagna. La costruzione geometrica adottata per questo comportamento considera un cilindro (nel caso di ambiente in 3D) o un rettangolo (in 2D), con lo scopo di valutare quando lo spazio considerato sia libero oppure ostruito da un ostacolo. Il cilindro e il rettangolo sono posizionati proprio davanti al veicolo con un diametro pari alla larghezza del personaggio, e di una lunghezza proporzionale alla velocità e all'agilità del veicolo.

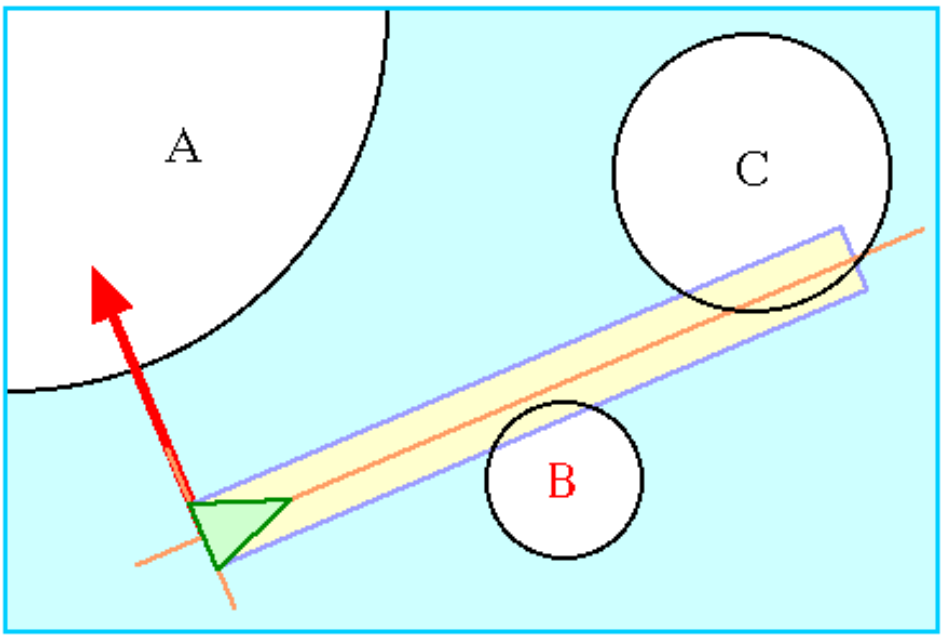

Figura 2.8: **Evitare ostacoli**

Quindi l'evasione dall'ostacolo considera se nello spazio del cilindro è presente qualche ostacolo della scena, determinando poi se si intersecano o meno. Una volta localizzato il centro di ogni ostacolo sferico, il test per valutare

l'intersezione è veloce. Il centro dell'ostacolo è proiettato sul piano verticale della scena, settando a zero la coordinata che non si trova sul piano considerato. Se la distanza dal punto al centro del veicolo risulta maggiore della somma del raggio dell'ostacolo e del veicolo, allora non c'è pericolo di collisione. L'ostacolo che interseca il cilindro del veicolo ed è più vicino, viene considerato come il più minaccioso. Nella figura 2.8 l'ostacolo A non interseca il cilindro, mentre gli ostacoli B e C si. Il più minaccioso è considerato l'ostacolo B, sarà quindi selezionato per l'elusione. In questo caso verrà applicata al veicolo una forza di sterzo verso sinistra. Se invece nessun ostacolo è in rotta di collisione, il valore restituito è un vettore nullo per indicare che per il momento non è necessaria una forza di sterzo correttiva. Da ricordare che nel caso in cui cooperino i comportamenti di ricerca e di evitare ostacoli, in genere ci si preoccupa soltanto degli ostacoli interposti tra l'agente e l'obbiettivo. Infatti nell'esempio dell'aereo, verrà ignorata una montagna che si trova alle spalle dell'aeroporto.

2.7.7 Wander

Wander è un tipo di comportamento casuale (figura 2.9). Una semplice implementazione del comportamento potrebbe essere realizzata generando una forza di guida random a ogni fotogramma, ma questo produrrebbe un movimento piuttosto particolare: un andamento "nervoso" e un movimento di svolta a tratti. Un approccio più interessante è quello di mantenere la direzione dello sterzo facendo piccoli spostamenti casuali in ogni fotogramma; in questo modo se il veicolo in un istante starà girando a destra, nel fotogramma successivo non cambierà bruscamente direzione ma manterrà quasi la stessa. In questo modo si ottiene una andatura casuale da una direzione all'altra. Altri comportamenti correlati a "vagare" sono **explore**, dove l'obiettivo è quello di trattare esaustivamente una determinata regione di spazio, e **forage**, che è una versione di vagare con l'obiettivo di cercare risorse.

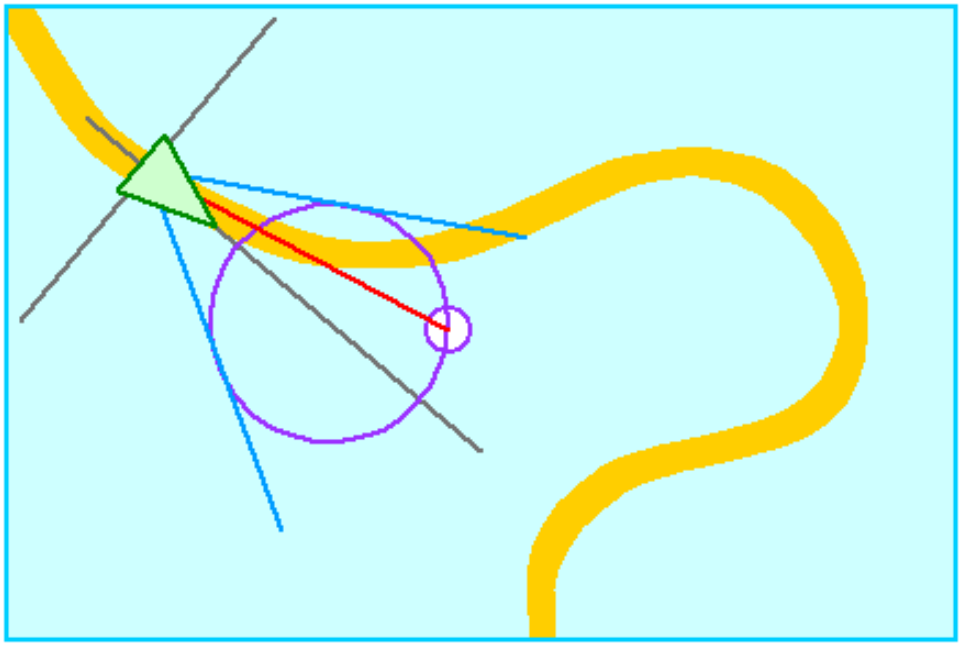

Figura 2.9: **Vagare**

2.7.8 Path following

Il comportamento **path following** consente a un personaggio di muoversi lungo un percorso prestabilito, come in un corridoio o in un tunnel. Questo comportamento non genera un vincolo rigido al percorso come potrebbe essere il movimento di un treno lungo un binario, ma rappresenta piuttosto il movimento delle automobili nel traffico; i percorsi individuali rimangono vicini e spesso paralleli alla linea bianca centrale della strada, ma non sono vincolati ad essa e sono liberi di deviare. Il corridoio è rappresentato da una spezzata (una serie di segmenti collegati) e un raggio che ne definisce la larghezza. L'obiettivo di tale comportamento è quello di far muovere l'agente lungo il percorso mantenendolo all'interno del raggio specificato. Se il veicolo è inizialmente lontano ed esterno al sentiero dovrà prima avvicinarsi ed entrare nello stesso. Per calcolare la forza di sterzo appropriata a questo comportamento è necessaria una previsione della posizione futura del veicolo in funzione della sua velocità. La posizione futura è proiettata sul punto più vicino della spezzata. Se la distanza di proiezione, dalla posizione prevista al punto più vicino del percorso, è inferiore al raggio che delimita il percorso allora il veicolo è considerato correttamente all'interno, e non

è necessaria una forza di governo correttiva. In tal caso viene quindi restituita come forza di guida un vettore nullo. Altrimenti il veicolo sta virando esternamente dal percorso oppure è lontano da esso ed è necessaria quindi una forza che lo riporti all'interno. In questo caso verrà utilizzato il comportamento di ricerca con obiettivo la proiezione della posizione futura sul percorso. Un percorso può essere seguito senza preoccuparsi del senso di marcia, oppure in una determinata direzione, regolando il punto di destinazione sul percorso nella direzione desiderata.

2.7.9 Flow field following

Il comportamento **flow field following** ovvero seguire campo di flusso fornisce uno strumento utile per indirizzare il movimento degli agenti in base alla loro posizione all'interno di un ambiente. Questo comportamento può essere molto interessante nel caso della produzione di videogiochi e nell'animazione perché consente di specificare il movimento da effettuare in base all'ambiente pianificando la scena. Il risultato di questo comportamento è che i personaggi allineano localmente il loro movimento in modo tangenziale rispetto al campo di forza (o campo di flusso). In questo caso viene definita una mappatura che può essere idealmente rappresentata da un piano con delle frecce disegnate, tale mappa riproduce la planimetria dell'ambiente. L'implementazione di questo comportamento viene realizzata stimando la posizione futura di un agente e valutando il campo di forza in quel punto. La direzione del flusso in quel punto rappresenta la direzione dello sterzo, mentre la velocità viene calcolata dalla differenza tra quella attuale e quella desiderata.

2.7.10 Unaligned collision avoidance

Il comportamento **unaligned collision avoidance** cerca di impedire la compenetrazione dei veicoli che si muovono in direzioni arbitrarie. Considerando l'esperienza di camminare in una folla all'interno di un atrio o una piazza, si sa che per prevenire le reciproche collisioni è necessario prevederle cambiando

direzione e velocità. Per implementare questo comportamento l'agente considera ognuno degli altri veicoli e determina in base alla velocità corrente quando e dove i due si potrebbero scontrare nel punto più vicino. La possibile collisione è determinata se il punto di scontro è previsto e se la distanza tra i veicoli nel punto di incontro è minore rispetto alla somma delle loro dimensioni. In questo caso sarà applicata una forza di sterzo laterale che guiderà gli agenti a evitare l'impatto allontanandosi dal punto di collisione previsto. Inoltre la forza li farà accelerare o decelerare in modo da raggiungere il punto della collisione prima o dopo il momento previsto per lo scontro (figura 2.10). Se i veicoli sono tutti allineati può essere utilizzata una strategia meno complicata attraverso il comportamento "separation".

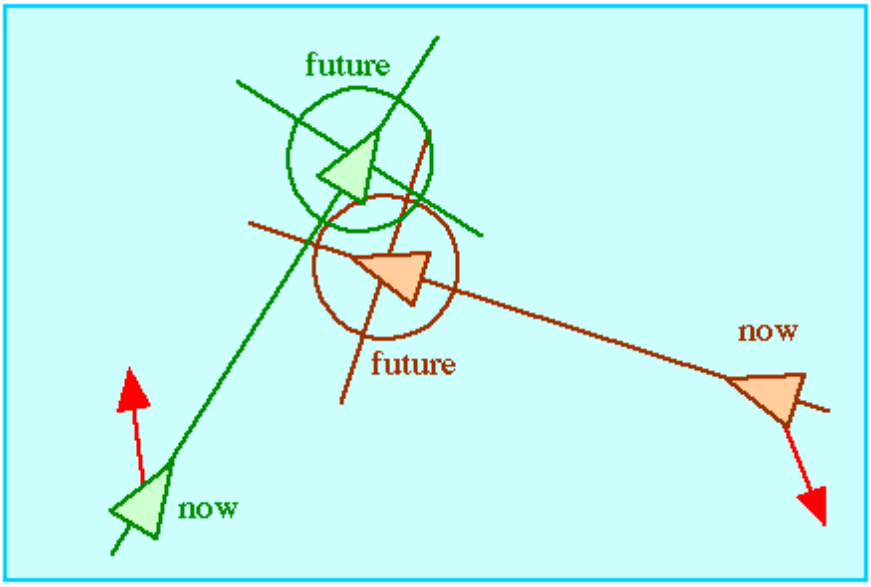

Figura 2.10: **Evitare le collisioni non allineati**

I prossimi tre comportamenti di guida: **separation**, **cohesion**, e **alignment**, (ovvero separazione, coesione e allineamento) si riferiscono a gruppi di agenti. In questi casi il comportamento determina come un personaggio reagisce agli altri agenti presenti nelle sue vicinanze. Prima di parlare di questi comportamenti è opportuno chiarire il concetto di "vicinanza". Il contesto locale di un agente è specificato da una distanza che determina quando due personaggi sono "vicini" e da un angolo che definisce il campo visivo percepito dall'agente. Gli agenti al di fuori del contesto locale di un veicolo vengono ignorati (Figura 2.11).

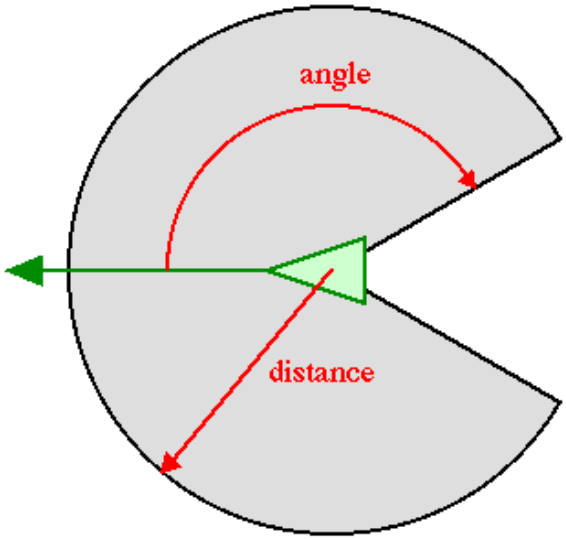

Figura 2.11: Concetto di **vicinanza**

2.7.11 Separation

Il comportamento **separation** dà all'agente la capacità di mantenere una certa distanza dagli altri nelle vicinanze. Esso può essere usato per prevenire le situazioni di affollamento. Per determinare la componente di separazione viene fatta prima una ricerca per trovare tutti i veicoli nelle vicinanze, e quindi viene calcolata la forza repulsiva. Sommando le forze repulsive calcolate per ogni personaggio nelle vicinanze si ottiene la forza di guida globale.

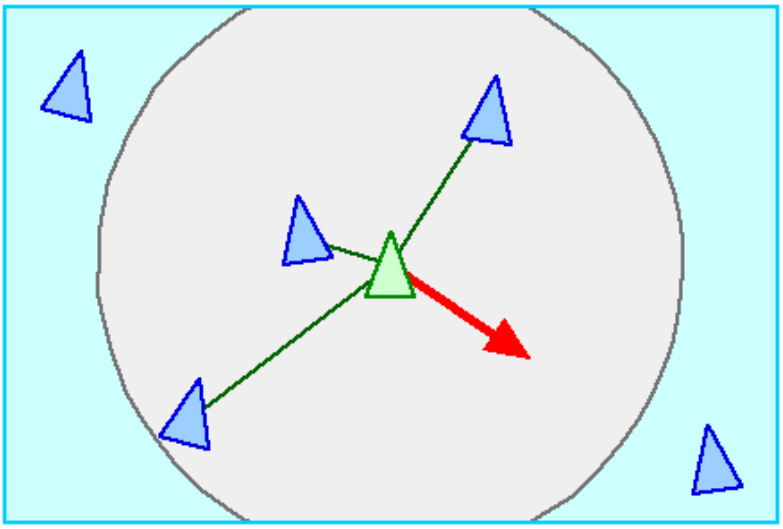

Figura 2.12: **Separazione**

2.7.12 Cohesion

Il comportamento **cohesion** serve per conferire al veicolo la capacità di avvicinarsi e formare un gruppo con altri agenti vicini (figura 2.13). Per implementare questo comportamento è necessario intanto fare una ricerca per determinare i personaggi che si trovano nelle vicinanze, calcolando poi la posizione media degli agenti. La forza di guida sarà applicata in direzione di questa posizione media oppure questo punto può essere usato come obiettivo del comportamento di ricerca.

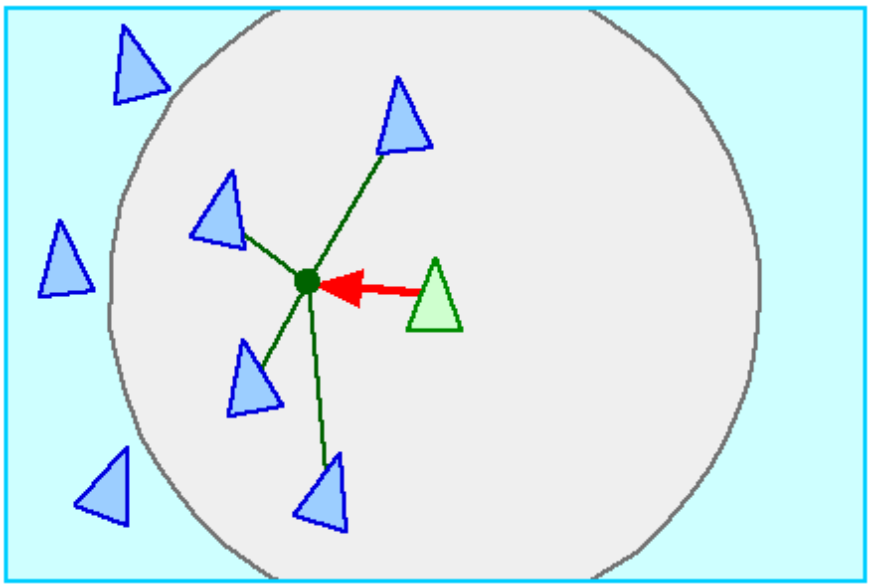

Figura 2.13: **la coesione**

2.7.13 Alignment

Il comportamento **alignment** dello sterzo consente all'agente di allinearsi con altri vicini rispetto a direzione e velocità (figura 2.14). Anche per questo comportamento è necessario prima determinare gli agenti che rientrano nel contesto locale dell'agente preso in considerazione e dopo si può calcolare la media delle loro velocità. Questa rappresenta la velocità desiderata, così il vettore di guida è calcolato dalla differenza tra la velocità media e quella attuale del veicolo. Questo vettore di sterzo tende a ruotare i personaggi allineandoli alla stessa direzione di quelli vicini.

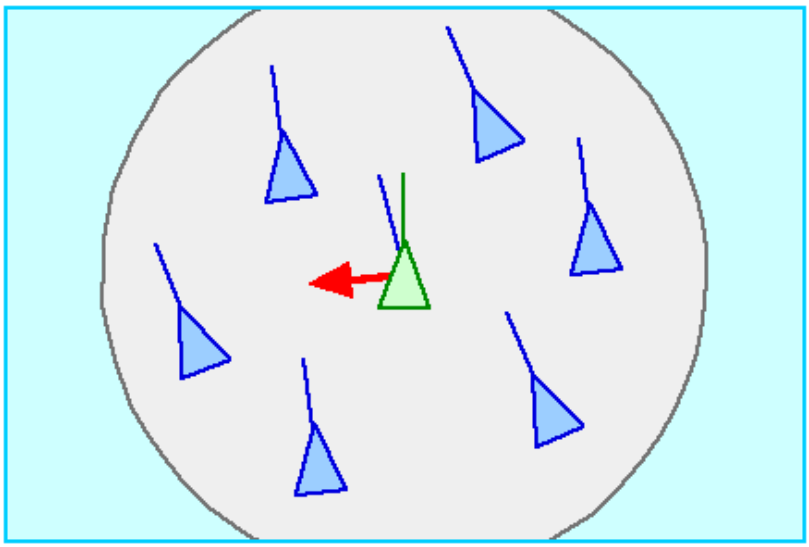

Figura 2.14: **Allineamento**

2.7.14 Flocking

Il comportamento **flocking** (affollamento) insieme agli altri comportamenti visti (**separation**, **cohesion** e **alignment**) può essere combinato per produrre un modello per greggi, eserciti e folle. In alcuni casi per produrre un unico sterzo che rappresenti l'affollamento basta semplicemente sommare le componenti dei tre diversi comportamenti utilizzati. Per ottenere però un migliore controllo sarebbe opportuno prima normalizzare le tre componenti di sterzo che si intende combinare, scalarle rispetto a tre fattori di ponderazione, e infine sommarle.

2.7.15 Leader following

Il comportamento **leader following** permette a uno o più agenti di seguirne un altro in movimento che viene designato come leader. Questo comportamento prevede che i seguaci vogliano stare vicini al leader senza affollare la sua traiettoria e avendo cura di non scontrarsi con esso. Inoltre se è presente più di un seguace, essi dovranno evitare di urtarsi tra di loro. L'implementazione di questo comportamento si basa sul comportamento arrival visto in precedenza (ovvero la ricerca di un punto, rallentando quando ci si avvicina ad esso).

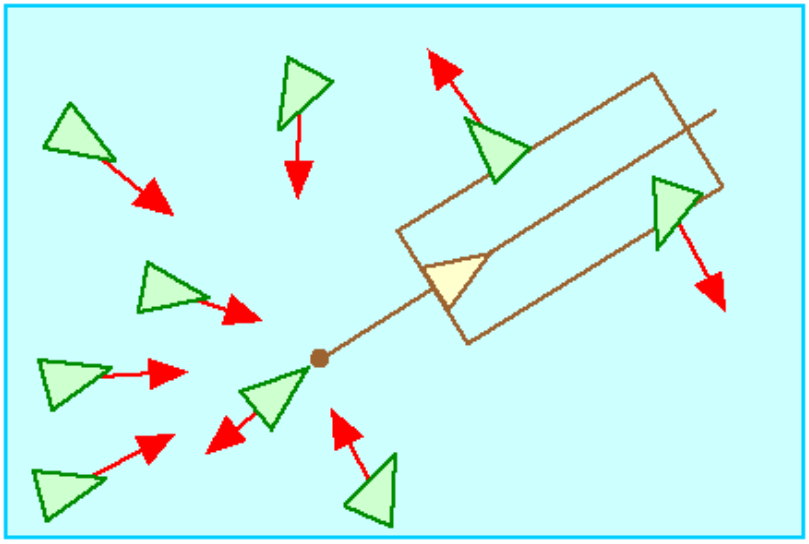

Figura 2.15: **Leader**

Il target di arrivo è un punto leggermente spostato dietro il leader, in modo da non superarlo (come mostrato in figura 2.15). Se nella regione rettangolare posta di fronte al leader è presente uno dei seguaci, questo sarà guidato lateralmente in modo da allontanarsi prima di ricominciare a cercare il target posto dietro al leader. Inoltre, i seguaci per evitare un sovraffollamento alle spalle del leader, adotteranno il comportamento separation visto prima.

2.7.16 Comportamenti combinati

I comportamenti di governo appena visti possono essere considerati componenti di una struttura più ampia ed essere utilizzati congiuntamente per la costruzione di comportamenti più complessi. Per rendere interessanti e realistici i comportamenti abbiamo bisogno di una fusione tra i singoli. Tale combinazione può avvenire in due modi:

1) Un personaggio può passare in modo sequenziale da un comportamento a un altro in seguito al sopraggiungere di particolari situazioni che si verificano all'interno del suo mondo.

2) I comportamenti possono essere fusi insieme nel senso che due o più forze di guida coesistono nello stesso momento.

Nel primo caso possiamo pensare a un gregge che pascolando in un prato venga attaccato da un branco di lupi. In questo modo sono cambiate le condizioni iniziali di quiete nel mondo-gregge e ciò attiva, come un interruttore, la variazione del comportamento: tutte le pecore del gregge non si preoccuperanno più di nutrirsi, ma cercheranno solo di fuggire dai lupi. Non c'è alcun senso nel mischiare i due comportamenti, perché certamente nessuna pecora può rallentare per continuare a nutrirsi, mentre è in fuga dai lupi.
Il cambiamento di stato comportamentale appena visto avviene nel livello di selezione azione della gerarchia a tre livelli.
Nel caso invece di fusione, ovvero quando due o più comportamenti si verificano in parallelo, avremo una situazione diversa dell'esempio appena visto. Mentre il gregge è in fuga dai lupi, ogni pecora deve scappare, ma anche evitare eventuali ostacoli come cespugli o alberi. Tutto questo deve essere fatto senza ignorare nessuno dei due comportamenti. In questo modo coesistono i comportamenti **evasion** e **obstacle avoidance**. Questa mescolanza di comportamenti avviene nel livello intermedio della gerarchia, ovvero lo sterzo.
La fusione dei comportamenti di guida può essere realizzato in diversi modi. Il più semplice è quello di calcolare ogni comportamento in modo indipendente e poi sommarli con un fattore di ponderazione per ciascuno di essi.
La combinazione lineare adottata per l'unione dei comportamenti spesso funziona bene, ma ha almeno due difetti: non è l'approccio più efficiente computazionalmente e, nonostante la somma pesata, le componenti possono annullarsi a vicenda in momenti inopportuni.
Per ridurre il carico di calcolo ripetuto ad ogni passo di simulazione si può pensare di valutare l'azione di ogni comportamento solo nel momento in cui questo intervenga.
Per risolvere il problema che le componenti si annullano a vicenda in particolari situazioni, può essere assegnata una priorità alle componenti del comportamento combinato. Ad esempio attribuisco a evitare ostacoli una priorità più alta rispetto all'evasione; innanzitutto verifico se il comportamento di evitare ostacoli è

necessario valutando se il valore del vettore di sterzo è diverso da zero, a questo punto se esiste il pericolo di collisione, si usa la componente appena stimata per evitarla. In caso contrario si verifica, allo stesso modo, il comportamento relativo alla seconda priorità, e così via.

Capitolo 3

Strumenti software utilizzati

3.1 Librerie Opensteer

OpenSteer (13) è una libreria open source di componenti realizzata per la costruzione di comportamenti di governo per i personaggi autonomi nei giochi ed altri tipi di simulazioni multi-agente. Solitamente gli agenti rappresentano personaggi (umani, animali, creature aliene), veicoli (auto, aerei, navicelle spaziali) o altri tipi di agenti mobili. Originariamente OpenSteer è stata progettata sul sistema operativo Linux e solo successivamente è stata sviluppata anche su Windows e Mac; viene distribuita come software open source con licenza MIT.

Questa libreria fornisce un insieme di comportamenti di sterzo per un agente mobile astratto chiamato *veicolo* e, allo scopo di consentire l'integrazione con i motori di gioco esistenti, può aggiungere nuove funzionalità attraverso eredità o stratificazione.

Oltre alla libreria, OpenSteer offre un'applicazione interattiva denominata OpenSteerDemo, che consente l'autoapprendimento di comportamenti di governo diversi; consente inoltre di creare comportamenti di sterzo e fornisce strumenti di visualizzazione. OpenSteerDemo è scritta in C + + e usa la grafica OpenGL. OpenSteerDemo si basa su un'architettura *plugin* in cui i plugin possono essere aggiunti in modo incrementale; esistono diversi plugin campione e chiunque può creare i propri partendo da questi. Nella creazione di un plugin occorre specificare

diverse azioni generiche richieste dal quadro OpenSteerDemo: aprire, chiudere, resettare, eseguire un passo di simulazione, visualizzare un fotogramma, e così via. Il plugin definisce le classi di veicoli e li gestisce in simulazione; esso definisce la geometria della scena in termini di ostacoli, percorsi e inoltre gestisce il controllo della telecamera e dei tasti funzione.

Nella versione OpenSteer 0.8.2, OpenSteerDemo permette all'utente di controllare in modo interattivo il tempo di simulazione (arresto, marcia, passo), selezionare il veicolo/personaggio, regolare la visualizzazione e monitorare il comportamento della telecamera.

OpenSteerDemo plugin fornisce al programmatore comportamenti prototipo di intelligenza artificiale in fase di progettazione del gioco, consentendone lo sviluppo prima che il motore principale del gioco sia completato. Dopo l'installazione si può lanciare l'applicazione OpenSteerDemo (su Linux, Mac o Windows è possibile farlo con un doppio clic su un'icona del desktop).

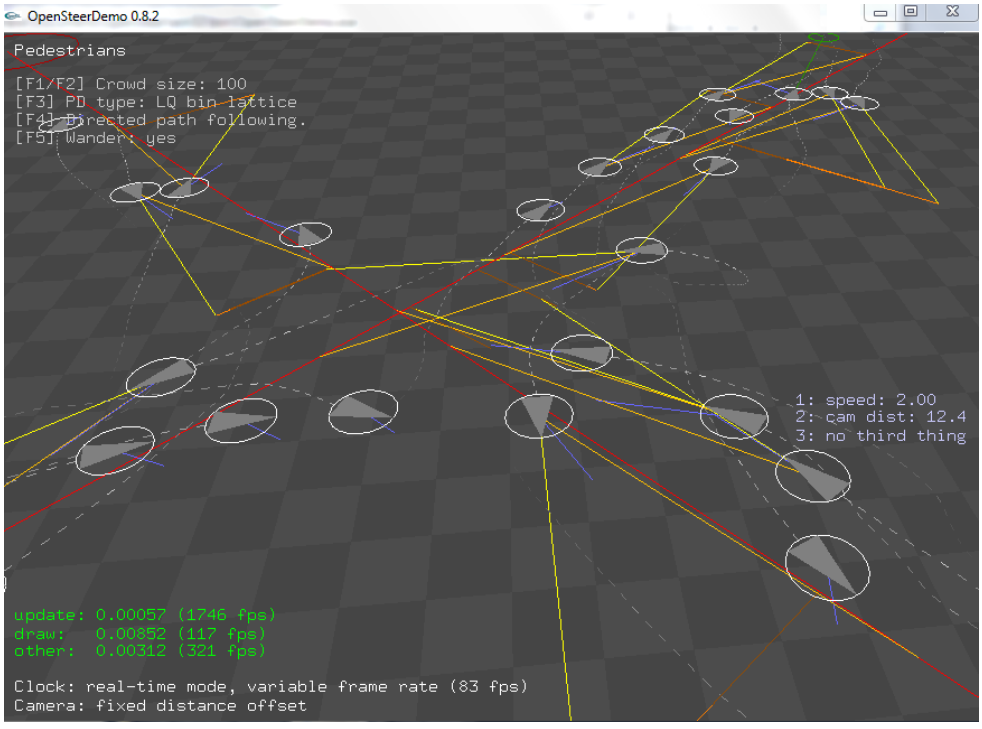

Figura 3.1: Tipica finestra OpenSteerDemo

Al primo utilizzo di OpenSteerDemo può risultare utile la guida di riferimento rapido. L'applicazione aprirà una finestra singola con etichetta "OpenSteerDemo 0.8.2" che mostrerà una schematica visualizzazione 3D del plugin di default. In

genere si vedranno veicoli in movimento con annotazioni testuali in sovrapposizione, come mostrato in figura 3.1.
OpenSteerDemo supporta l'interazione basata sul movimento e click del mouse per la vista e regolazione della telecamera, e selezione del veicolo; inoltre riconosce alcuni singoli comandi fondamentali da tastiera:

- Tab, seleziona il prossimo plugin;
- R, riavvia il plugin corrente;
- S, seleziona il prossimo veicolo;
- C, seleziona altra modalità telecamera;
- F, seleziona il tasso di frame preimpostato (24fps, 60fps);
- Space, passa da esegui a pausa;
- Freccia destra, avanza di un fotogramma;
- ?, stampa a video la guida di riferimento rapido;
- Esc, esce da OpenSteerDemo

Un plugin può consentire la gestione dei tasti funzione (da F1 a F12 nella maggior parte delle tastiere). È possibile digitare "**?**" per stampare un elenco di tutti i comandi riconosciuti, compresi quelli gestiti dal plugin corrente.
OpenSteerDemo tiene traccia del movimento di un singolo agente *scelto* che sarà seguito dalla telecamera; in alcune demo sono previste annotazioni aggiuntive per l'agente scelto. L'utente può modificare la selezione con il mouse: puntando il mouse in prossimità di un agente lo farà evidenziare; facendo clic sull'agente (col tasto sinistro del mouse) questo verrà selezionato.
La telecamera di OpenSteerDemo (ovvero il *punto di vista*) si regola automaticamente per mantenere in vista il veicolo scelto; esistono differenti modi di visualizzazione di una scena in funzione del tipo di telecamera scelta:

- statico: la telecamera è fissa;
- distanza fissa di offset: la telecamera rimane a distanza fissa dal veicolo selezionato;
- locale offset fisso: la telecamera rimane fissa nello spazio in prossimità del veicolo selezionato;
- verso il basso: la telecamera guarda il veicolo scelto direttamente dall'alto;

- offset POV: la telecamera guarda in avanti rispetto al veicolo scelto con un determinato offset nello spazio locale del veicolo.

È possibile regolare il punto di vista della telecamera con il mouse (o qualsiasi altro dispositivo di input). OpenSteerDemo consente di regolare la posizione della telecamera rispetto al suo obiettivo prevedendone la posizione futura. Per regolare la telecamera occorre tenere premuto il tasto Ctrl mentre si trascina il mouse con il tasto sinistro premuto. Il movimento 2D del mouse è mappato all'interno dello spazio della telecamera, ruotando la posizione della stessa attorno al suo obiettivo; questo vincola la telecamera a muoversi sulla superficie di una sfera centrata nell'obiettivo; il raggio della sfera (la "distanza di visione") può essere regolato trascinando il mouse con il tasto centrale premuto. Se si utilizza un mouse senza tasto centrale è possibile regolare il raggio tenendo premuti contemporaneamente i tasti Ctrl e Alt mentre si trascina il mouse con il tasto sinistro premuto.
Per mantenere una visione stabile la telecamera presente in OpenSteerDemo effettuerà una interpolazione delle successive posizioni occupate dal veicolo scelto, tenendo anche conto delle regolazioni operate attraverso il mouse.
L'orologio OpenSteerDemo fornisce due dati: quello del *tempo reale* e quello del *tempo di simulazione*. L'aggiornamento dell'immagine della telecamera si basa sul tempo reale; il tempo di simulazione normalmente segue il tempo reale ma può anche essere *messo in pausa*; in tal caso la simulazione congela il movimento del veicolo. Questa possibilità consente l'esame di dettagli e di riposizionare in modo interattivo la telecamera. Nella attuale implementazione, quando il tempo di simulazione è in pausa, gli aggiornamenti di simulazione continuano ad andare avanti a ogni frame, ma il *tempo trascorso* (dt) è indicato pari a zero. L'orologio può essere impostato per funzionare in una delle tre modalità descritte:

- variable frame rate, per la visualizzazione delle simulazioni in tempo reale, simile alla grafica su un PC;
- fixed target frame rate, la visualizzazione procede a multipli di un frame rate fisso, simile alla grafica su una console per videogiochi;
- animation mode, per l'esecuzione di simulazioni non in tempo reale; la visualizzazione appare in "slow motion".

3.1.1 Plugin campione

OpenSteerDemo è distribuito con diversi plugin campione, utili sia come dimostrazioni di comportamenti di governo che come codice di esempio. I più significativi plugin campione sono descritti di seguito.

Capture the Flag: Questo plugin è stato proposto da M. Chady del gruppo di lavoro IGDA di AI Interface Standards Committee. Un singolo agente cercatore o attaccante tenta di raggiungere un obiettivo centrale mentre quattro difensori tentano di intercettare l'attaccante prima che raggiunga l'obiettivo (figura 3.2). Il campo di gioco è disseminato di ostacoli. Gli ostacoli sono rappresentati da sfere. Premendo F1 si può aggiungere un ostacolo (fino a 100), mentre con F2 se ne rimuove uno. Il "gioco" finisce quando l'attaccante raggiunge l'obiettivo o un difensore riesce a intercettare l'attaccante. La demo si riavvia automaticamente dopo pochi secondi.

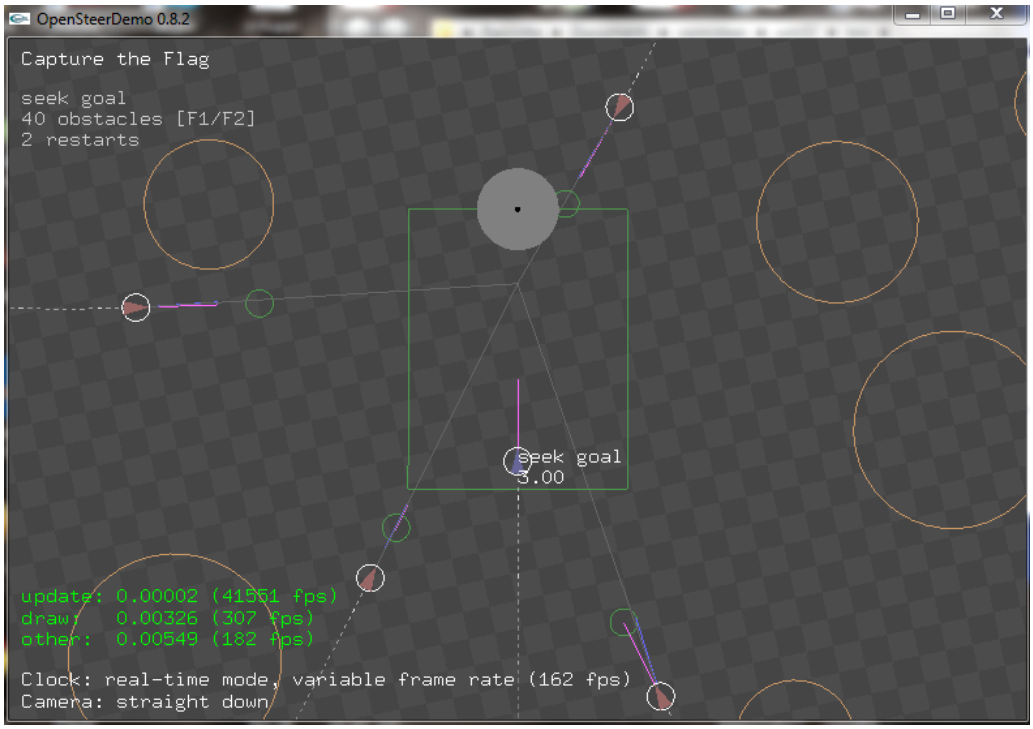

Figura 3.2: Screenshot del plugin Capture the flag

L'attaccante combina tre comportamenti di guida fondamentali: la *ricerca di un obiettivo*, *l'evasione* e *l'elusione dell'ostacolo*; i difensori combinano i

comportamenti di *ricerca* ed *evitare ostacoli*. Un modo per realizzare il comportamento non banale all'attaccante è quello di rilevare quando un difensore entra nel "corridoio" tra l'attaccante stesso e la meta. Questo "corridoio" è raffigurato da un contorno verde e rappresenta una sezione libera che porterà l'attaccante a raggiungere la meta. Se il corridoio è libero da difensori l'attaccante utilizza solo il comportamento *cercare* tralasciando l'*elusione* dei difensori nelle vicinanze.

Il comportamento di ricerca dei difensori non prevede la conoscenza a priori del percorso dell'attaccante o della meta che dovrà raggiungere. Pertanto i difensori non adottano una strategia globale e, per esempio, non cercano di porsi tra l'attaccante e l'obiettivo.

I difensori si ignorano a vicenda ma, per un possibile miglioramento del plugin, dovrebbero evitare le collisioni l'uno con l'altro e coordinare il loro inseguimento. L'attaccante è confuso quando più difensori convergono da direzioni diverse. Gli ostacoli vengono sempre evitati impostando una direzione tangenziale rispetto al bordo più vicino che non sempre è quella ottimale considerando gli obiettivi complessivi dell'attaccante.

MapDrive[4]: questa demo mostra un veicolo che si muove all'interno di un percorso definito come una serie di punti di larghezza fissata e disseminato di ostacoli (figura 3.3). Il veicolo dispone di sensori che creano una mappa in funzione del terreno circostante classificandolo come percorribile o non percorribile. Il veicolo cerca di seguire il percorso, evitando gli ostacoli e massimizzando la velocità. Quando il veicolo si trova in pericolo di collisione cerca di evitare gli ostacoli e, se impossibilitato, si "arrende" diventando giallo e rallentando fino a fermarsi. Se si scontra con un ostacolo, il veicolo diventa rosso. In entrambi i casi la simulazione si riavvia.

[4] Questo plugin è ispirato alla DARPA Grand Challenge, gara di cross country per veicoli autonomi.

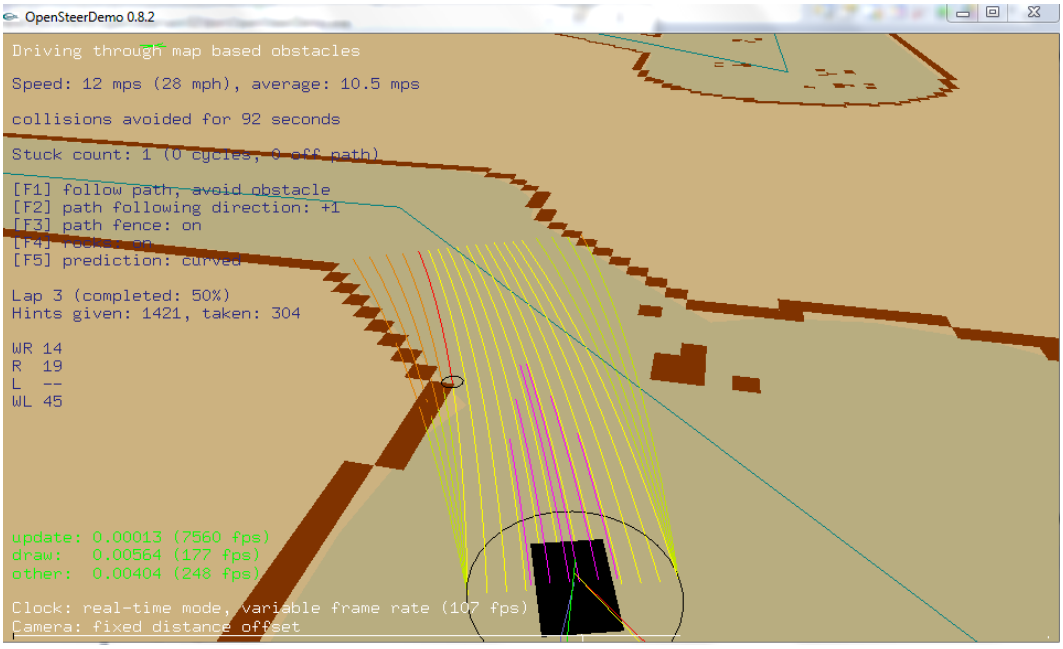

Figura 3.3: Screenshot tratto dal plugin Mapdrive

Pedestrians: 100 agenti seguono un percorso costituito da una linea rossa cercando di non scontrarsi tra loro ed evitando due grandi ostacoli sferici posizionati lungo il percorso. È possibile durante la simulazione aggiungere un pedone (premendo F1) o rimuoverlo (con F2). Questa simulazione multiagente utilizza un database spaziale per accelerare le query di prossimità; F3 permette di cambiare il tipo di database. Quando si raggiunge la fine del percorso i pedoni invertono la loro direzione.

Boids: 200 agenti simulano il volo di uno stormo di uccelli. Tale plugin realizza una combinazione dei comportamenti di guida di *separazione*, *allineamento* e *coesione*. Durante la simulazione si può premere F1 per aggiungere un agente o F2 per rimuoverlo. Questa simulazione multiagente utilizza un database spaziale per accelerare le query di prossimità; F3 permette di cambiare il tipo del database di prossimità. Lo stormo vola all'interno di una sfera. Quando gli agenti si trovano sulle parti estreme della sfera, si possono verificare due situazioni differenti: nel caso di "steer back when outside" un agente che si trovi fuori della sfera viene attirato verso il centro; mentre con "wrap around teleport" gli agenti che volano fuori dalla sfera sono immediatamente riposizionati sul lato opposto agli antipodi. Premendo F4 si può scegliere una delle due alternative.

Soccer[5]**:** in questo caso viene simulata una partita di calcio (figura 3.4). Si compone di due squadre (blu e rossa) di 8 giocatori ciascuna, un pallone verde e un rettangolo di gioco utilizzato per rappresentare il campo e le due porte. Mentre il gioco procede il plugin aggiorna il punteggio delle due squadre. Il pallone è derivato da SimpleVehicle ma per il suo movimento sono applicate le leggi fisiche che regolano il rimbalzo.

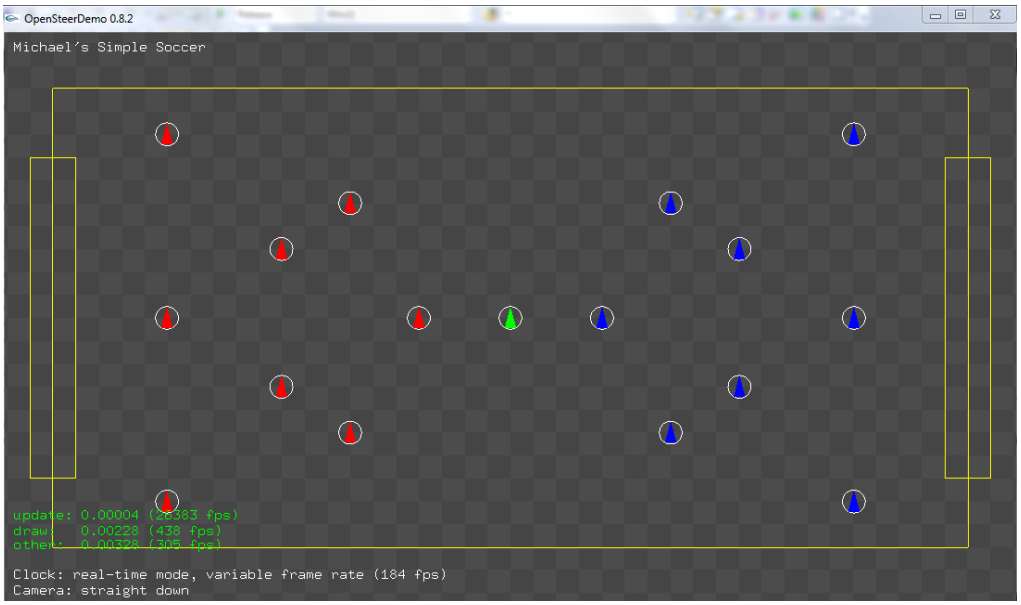

Figura 3.4: Screenshot tratto dal plugin Soccer

Multiple Pursuit: Consiste in un dispositivo di prova per l'utilizzo del comportamento *cercare*. Diversi agenti ne inseguono un altro a cui è stato applicato il comportamento "vagare". Le linee colorate indicano la futura posizione prevista della preda utilizzata come bersaglio dagli inseguitori. Quando un inseguitore raggiunge la preda esso è riposizionato via, in una nuova posizione casuale.

Low Speed Turn: un dispositivo di prova utile a valutare la risposta di un veicolo a cui viene applicata una forza di sterzo rivolta all'indietro mentre si muove a bassa velocità.

[5] è un contributo di Michael Holm di IO Interactive, il 9 luglio 2003.

One Turning Away: questo è stato il primo plugin realizzato, ora serve come esempio minimale.

3.1.2 Organizzazione a plugin

I comportamenti di guida rappresentano un tipo particolare di programmazione. Uniscono creatività al rigore matematico della programmazione di algoritmi. OpenSteer permette di concentrarsi sullo sviluppo di comportamenti di guida senza doversi preoccupare di altro. Ciò è consentito dal meccanismo dei plugin come gli esempi presenti nella OpenSteerDemo. Quando si scrive un plugin, è necessario fornire solo il codice che è specifico per il veicolo e per il comportamento di guida.
Nelle librerie il file che contiene informazioni dettagliate sull'interfaccia plugin in C + + è **PlugIn.h** che definisce due classi: `AbstractPlugIn` (una classe pura, o interfaccia virtuale) e `PlugIn` che prevede l'implementazione di base. Per creare un nuovo plugin si deve definire una nuova classe che eredita l'implementazione di base da `PlugIn`. Esso fornisce alcuni metodi di default che è possibile personalizzare. È necessario implementare sei metodi per la nuova classe per soddisfare la struttura di `AbstractPlugIn`:

- name, restituisce un nome di stringa di caratteri;
- open, consente di allocare e inizializzare i veicoli;
- close, permette di deallocare i veicoli;
- update, esegue un passo della simulazione;
- redraw, aggiorna tutta la grafica per fotogramma;
- allVehicles, restituisce un unico gruppo con tutti i veicoli di questo plugin.

Per un esempio minimale di plugin si può vedere OneTurningAway (**opensteer/plugins/OneTurning.cpp**), che definisce un unico veicolo con comportamento banale. LowSpeedTurn è un altro esempio di plugin con comportamenti elementari ma agisce con più veicoli. Pedestrian e CaptureTheFlag sono i due esempi più complessi.
Il modo più semplice per iniziare la realizzazione di un plugin è:

1) prendere uno dei plugin campione più strettamente correlato a quello che si desidera realizzare;
2) farne una copia rinominata;
3) aggiungerlo in OpenSteerDemo;
4) verificare che il nuovo plugin appaia e abbia il comportamento originario,
5) apportare modifiche incrementali al nuovo plugin.

Se si utilizza un IDE, ambiente di sviluppo integrato come Visual Studio, si dovrà aggiungere il nuovo file all'interno del progetto.
Nella versione di Visual Studio 2010 (utilizzata per questo lavoro di tesi) basta fare click col tasto destro del mouse sulla cartella plugin all'interno dell'elenco progetti ed aggiungere, come nuovo elemento, un file .cpp.
Ogni esempio di plugin presente nella Demo in **opensteer/plugins/** definisce una nuova classe veicolo e una nuova classe PlugIn. Di solito i plugin contengono una o più categorie di autoveicoli; OpenSteer cerca di fornire una grande flessibilità al modo in cui questo può essere fatto. L'approccio più semplice, caratteristico di tutti gli esempi forniti, è di definire una nuova classe di veicoli ereditando da `SimpleVehicle`.
OpenSteerDemo e i relativi plugin utilizzano due tipi definiti in `AbstractVehicle`: `AVGroup` e `AVIterator`.

- `AVGroup` viene utilizzato ogni volta che un gruppo di `AbstractVehicle` ha bisogno di essere passato come argomento a una funzione.
- `AVIterator` viene utilizzato per scorrere il contenuto di un `AVGroup`. Questi tipi sono basati sulle STL (Standard Template Library) del C + +.

`AVGroup` e `AVIterator` vengono utilizzati all'interno della biblioteca OpenSteer ma, volendo evitare di utilizzare le STL, si può facilmente sostituirli con la propria implementazione.
Per scrivere un plugin occorre ricordare:

- I tempi sono misurati in secondi e passati come valori float.

- `Vec3` è il tipo usato in tutto OpenSteer per rappresentare vettori geometrici nello spazio 3D cartesiano. I colori sono attualmente rappresentati come dati di tipo `Vec3`.
- Per agevolare il debug o l'analisi del comportamento dello sterzo è consigliabile fornire ulteriori informazioni relative al veicolo utilizzando annotazioni grafiche o di stampa.

3.1.3 Organizzazione e struttura delle classi

La struttura delle classi OpenSteer è stata progettata per garantire flessibilità e rendere agevole l'integrazione con codice già esistente.
Le classi sono state progettate per consentire di ereditare le loro funzionalità, oppure scalarle in cima a quelle delle classi esistenti; quest'ultima possibilità è supportata dal concetto di classi mixin (una classe con superclasse template). La realizzazione di questo concetto può essere ottenuta con l'ereditarietà multipla delle classi di base preesistenti e le classi di OpenSteer, ma la maggior parte dei programmatori C + + preferisce evitare l'ereditarietà multipla.
Molte delle classi di OpenSteer sono definite in tre parti:

1) un protocollo astratto (interfaccia o classe virtuale pura: "AbstractLocalSpace");
2) un'implementazione espressa in un mix "LocalSpaceMixin";
3) una classe istanziabile realizzata attraverso la sovrapposizione "LocalSpace".

Nell'organizzazione delle librerie OpenSteer, il gruppo principale di comportamenti di guida è contenuto nella classe SteerLibraryMixin definito in **SteerLibrary.h**. Un mixin è una classe con superclasse template basata su modelli e viene utilizzata per aggiungere un insieme di metodi per una determinata classe base. In questo caso SteerLibraryMixin aggiunge funzionalità a una classe che supporta l'interfaccia AbstractVehicle. Ad esempio, SimpleVehicle combina SteerLibraryMixin e diverse altre utility con AbstractVehicle. Come risultato SimpleVehicle ha tutti i metodi definiti in SteerLibraryMixin.
Esistono dei metodi ausiliari, definiti e contenuti in SteerLibraryMixin. Sono chiamati quando entrano in azione particolari comportamenti di guida. Questi

metodi sono destinati ad essere molto usati nei plugin per fornire annotazione grafica, o altri effetti secondari.
Molti comportamenti di guida potrebbero restituire un valore zero, ovvero un vettore con valori (0, 0, 0), per indicare che tale comportamento non è necessario in quel momento, solitamente perché l'obiettivo è già stato raggiunto.
I comportamenti contenuti nella classe SteerLibraryMixin sono:

comportamento Wander (vagare)

Vec3 steerForWander (float dt);
Restituisce una forza di governo per il comportamento passeggiata.

comportamento Seek (cercare)

Vec3 steerForSeek (const Vec3& target);
Restituisce una forza di governo per cercare il percorso verso la destinazione specificata. Provoca la variazione di direzione del veicolo verso il bersaglio e il suo spostamento verso di esso.

comportamento Flee (fuggire)

Vec3 steerForFlee (const Vec3& target);
Restituisce una forza di governo per allontanarsi dal luogo specificato. L'applicazione di questa forza provoca nel veicolo un cambio di direzione, che diventa opposta a quella del bersaglio, e l'allontanamento da esso.

comportamento Path Following

Vec3 steerToFollowPath (const int direction, const float predictionTime, Pathway& path)
Vec3 steerToStayOnPath (const float predictionTime, Pathway& path)
Restituisce una forza di governo che permette al veicolo di seguire un percorso specificato. Esistono due varianti: `steerToStayOnPath` cerca solo di mantenere il veicolo sul percorso; `steerToFollowPath` fornisce l'indicazione del percorso su

cui il veicolo deve rimanere; inoltre il veicolo deve seguire il percorso nel verso indicato dall'argomento `direction`, che può essere +1 o -1.
La forza di guida è determinata sulla base di una previsione della posizione futura del veicolo, attraverso `predictionTime`. Se tale posizione prevista è all'interno del percorso (e, nel caso di `steerToFollowPath`, è anche nella giusta direzione), questa funzione restituisce un valore zero del vettore. Altrimenti la funzione restituisce una forza di guida che permette di dirigere il veicolo verso un punto del percorso.

comportamento Obstacle Avoidance

Vec3 steerToAvoidObstacle (const float minTimeToCollision, const Obstacle& obstacle);
Vec3 steerToAvoidObstacles (const float minTimeToCollision, const ObstacleGroup& obstacles)
Restituisce una forza di governo per evitare ostacoli che possono essere singoli o specificati come gruppo, ovvero un ObstacleGroup (STL un vettore di puntatori ostacolo). La forza di guida sarà puramente laterale e permetterà al veicolo di muoversi costeggiando il bordo laterale dell'ostacolo. È necessario evitare l'ostacolo se:

1) interseca l'attuale percorso del veicolo,
2) è di fronte al veicolo,
3) si trova a `minTimeToCollision` secondi di distanza, in base all'attuale velocità del veicolo.

Se sono stati specificati molti ostacoli ed esistono molteplici potenziali collisioni verrà preso in considerazione il più vicino. Quando non è richiesto di evitare un ostacolo questa funzione restituisce un valore di vettore pari a zero.

comportamento Unaligned Collision Avoidance

Vec3 steerToAvoidNeighbors (const float minTimeToCollision, const AVGroup& others);
Restituisce una forza di governo per evitare la collisione con altri veicoli nelle vicinanze che si muovono in direzioni casuali. In primo luogo si deve determinare quali altri veicoli sono in rotta di collisione con il primo, successivamente si applica la forza per evitare la potenziale collisione. L'attuale versione permette di

sterzare solo lateralmente, non accelerare né rallentare. La funzione restituisce un vettore forza di governo pari a zero se non vi è collisione imminente.

comportamento Separation

Vec3 steerForSeparation (const float maxDistance, const float cosMaxAngle, const AVGroup& flock);

Restituisce una forza di governo per far muovere il veicolo lontano dagli altri nelle vicinanze.

comportamento Alignment

Vec3 steerForAlignment (const float maxDistance, const float cosMaxAngle, const AVGroup& flock);

Restituisce una forza di governo che permette al veicolo di allineare la propria direzione con quella degli altri nelle vicinanze.

comportamento Cohesion

Vec3 steerForCohesion (const float maxDistance, const float cosMaxAngle, const AVGroup& flock);

Restituisce una forza di governo per far muovere il veicolo in direzione del centro di massa degli altri nelle vicinanze.

comportamento Pursuit

Vec3 steerForPursuit (const AbstractVehicle& quarry);

Vec3 steerForPursuit (const AbstractVehicle& quarry, const float maxPredictionTime);

Restituisce una forza di governo che permette di cercare un altro veicolo in movimento. L'agente si dirige verso il punto previsto di intercettazione.

comportamento Evasion

Vec3 steerForEvasion (const AbstractVehicle& menace, const float maxPredictionTime);

Restituisce una forza di governo in grado di evitare un altro veicolo in movimento. L'agente si dirige lontano dal punto previsto di intercettazione.

comportamento Speed Maintenance

Vec3 steerForTargetSpeed (const float targetSpeed);
Restituisce una forza di governo per mantenere una velocità assegnata. Il valore sarà inteso in senso longitudinale rispetto all'asse del veicolo e la sua lunghezza sarà troncata al paramentro maxForce.

3.2 Ambiente grafico per simulazione di folle

Per rendere visivamente più interessante il presente lavoro si è pensato di integrarlo con quello realizzato all'interno dei laboratori dell'università in un precedente progetto di tesi. Quest'ultimo prevedeva la simulazione di folle all'interno di un ambiente basato su Horde3D (14).

3.2.1 Horde3D

Horde3D (figura 3.5) è un applicativo software open source realizzato dall'università di Asburgo che permette di realizzare simulazioni grafiche per l'animazione e il rendering 3D e possiede potenzialità che non hanno nulla da invidiare a prodotti commerciali. Horde3D supporta diversi linguaggi di programmazione, tra cui C#, Java e altri. Inoltre permette di lavorare su Windows, Linux e Mac OS X. Questo motore grafico tratta diversi oggetti (luci, materiali, modelli) messi a disposizione dell'utente attraverso gli handles (gestori simili a puntatori); gli handles sono memorizzati dall'applicazione al fine di renderne possibile l'utilizzo, la modifica delle proprietà e il rilascio della memoria.
Il punto di forza di Horde3D è il rendering di buona qualità anche per folle di grandi dimensioni. Gli aspetti che rendono ciò possibile sono:

- i dati relativi alle risorse grafiche sono memorizzati internamente per agevolare la velocità delle animazioni,
- la geometria è stata ottimizzata al fine di ottenere la massima efficienza dall'uso della cache,
- si può scegliere tra diversi livelli di dettaglio per i modelli,

- si può velocizzare il rendering di scene con molte luci utilizzando diverse tecniche (vertex skinning e deferred shading).

Figura 3.5: Schermata relativa alla Demo di Horde3D

3.2.2 ColladaConverter

Horde3D utilizza il ColladaConverter, (COLLAborative Design Activity) formato standard su file XML creato da Sony Entertainment i cui file sorgente sono oggi distribuiti gratuitamente. Il formato Collada (15) non rappresenta un formato finale per modelli, ma è fondamentale per importare dall'esterno le risorse software per la modellazione 3D (come mesh poligonali e le altre caratteristiche utili a definire la scena); ColladaConverter consente di convertire nel formato adatto ogni oggetto realizzato con i diversi strumenti di modellazione. In particolare ColladaConverter permette di trasformare il formato Collada 1.4 in un formato riconosciuto da Horde3D. Per il progetto dell'ambiente grafico di simulazione di folle è stato necessario esportare in formato Collada sia le abitazioni e gli edifici realizzati con Google SketchUp Pro (16), che i modelli di persone e l'animazione realizzati con 3ds Max 9 (17); per quest'ultimo è stato necessario utilizzare un apposito tool di conversione chiamato ColladaMax (18).

3.2.3 Google SketchUp Pro

Google SketchUp Pro è un applicativo potente e semplice da usare che permette di realizzare forme bidimensionali e tridimensionali; è stato impiegato per la creazione di elementi architettonici (con un basso numero di poligoni) presenti nello scenario del simulatore ed esportarli nel formato adatto a Horde3D (.dae). SketchUp permette anche di scaricare gratuitamente modelli di qualsiasi genere messi a disposizione su un archivio online.

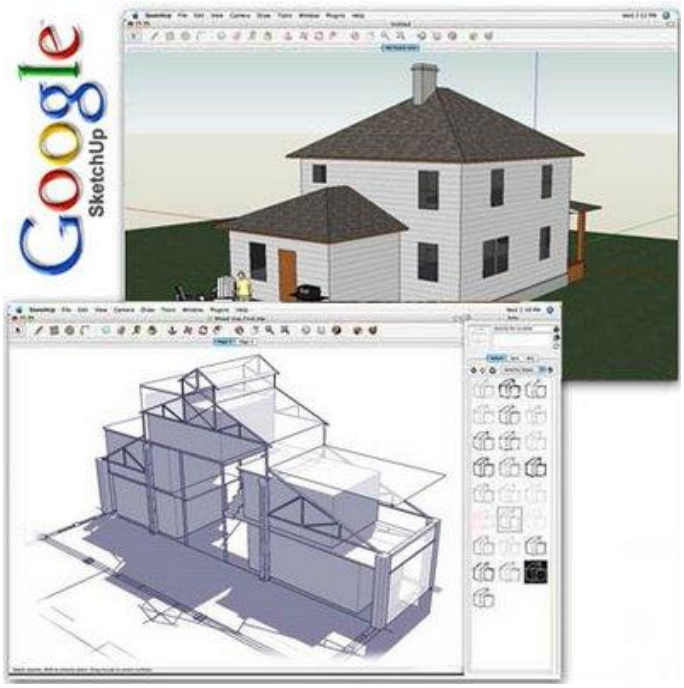

Figura 3.6: Schermata di lavoro di Google SketchUp

3.2.4 Autodesk 3ds Max 9

3ds Max 9 è un programma di Autodesk per la grafica vettoriale tridimensionale e animazione che permette la realizzazione di modelli 3D. Grazie al plugin ColladaMax è stato possibile dotare di uno scheletro interno i modelli umani, animarli ed esportarli in un formato riconosciuto da Horde 3D (.dae).

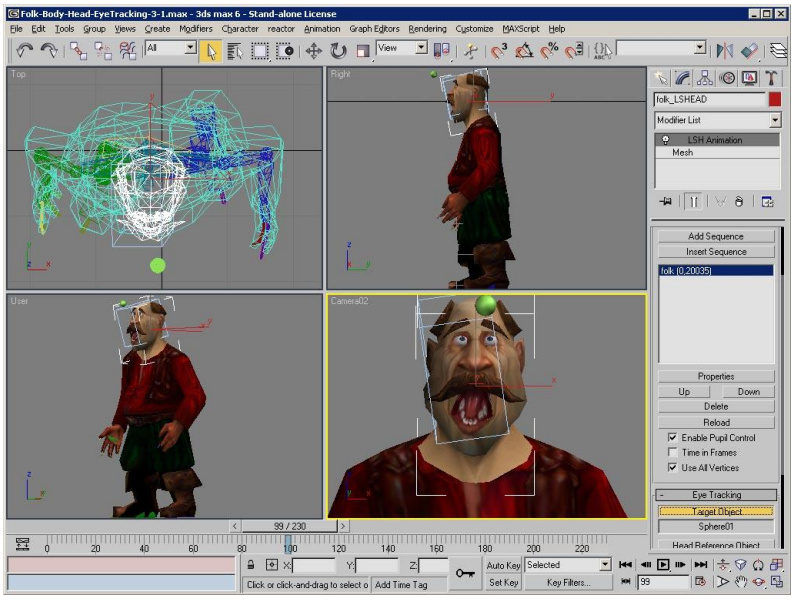

Figura 3.7: Immagine di una schermata d'esempio di 3dsMax

3.2.5 CubeMapGen

Nel simulatore grafico è stato inserito uno scenario come panorama. Per costruirlo è stato usato uno skybox, un metodo utilizzato nei videogiochi per far sembrare l'ambiente circostante più grande. Uno skybox si ottiene disegnando sulle facce di un cubo le texture del cielo, delle montagne e del piano su cui si muovono gli oggetti, mappandoli in modo opportuno come mostrato in figura 3.8 e 3.9.

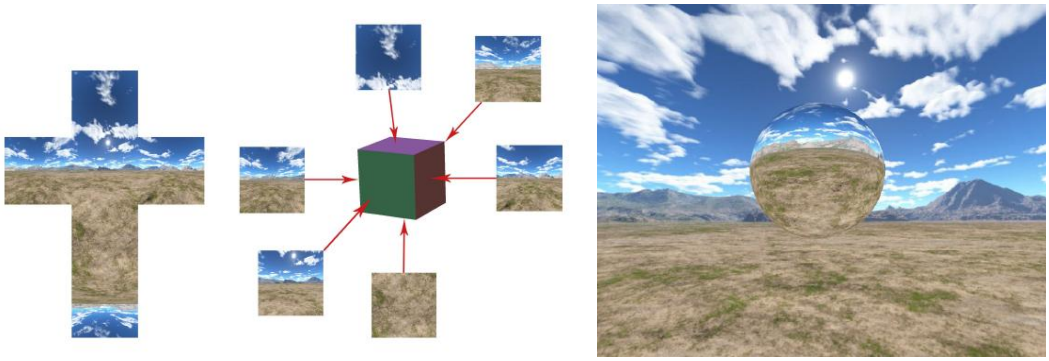

Figura 3.8: Cubemap e skybox utilizzati nel sistema di folle 3D

Il risultato sarà una cubemap che darà l'impressione di essere circondati da entità tridimensionali in lontananza. Per realizzare la cubemap si è sfruttato

CubeMapGen, un software freewere che permette di importare, creare, visualizzare ed esportare cubemap.

Figura 3.9: Esempio di panorama mappato sulle facce di un cubo

Per ulteriori dettagli sull'ambiente grafico 3D finalizzato alla simulazione di folle si rimanda al lavoro di tesi (19).

Capitolo 4
Il sistema realizzato

4.1 Descrizione del sistema

L'obiettivo del presente lavoro di tesi consiste nell'affrontare e trovare soluzione ad alcuni problemi di simulazione di folle riscontrati in un precedente progetto di tesi realizzato presso il Laboratorio di Robotica del Dipartimento di Ingegneria dell'Informazione. Tale lavoro aveva lasciato intravedere possibili sviluppi futuri, come la realizzazione di un sistema di "collision detection" per gli ostacoli presenti nella scena. Ci si è pertanto posto l'obiettivo di combinare comportamenti diversi messi a disposizione dalle librerie Opensteer, ed è stato sviluppato un nuovo sistema capace di evitare che i pedoni entrino in collisione tra di loro o con gli ostacoli presenti, prestando particolare attenzione alla pianificazione delle traiettorie.

Dopo un'analisi dei comportamenti messi a disposizione dalle librerie esistenti di Opensteer, l'obiettivo è stato quello di combinare comportamenti diversi in modo che fossero applicati in parallelo e contemporaneamente agli agenti nella scena. Inizialmente non ci si è preoccupati dell'aspetto estetico dell'applicazione e si è condotto il progetto appoggiandosi alla grafica utilizzata dalle librerie Opensteer. Tale grafica è molto semplice e prevede che gli agenti siano rappresentati come

piccoli cerchi con inscritto un triangolo (che ne indica l'orientamento), mentre l'immagine degli ostacoli è rappresentata da semplici cerchi di dimensione e colore variabili; gli obiettivi o target degli agenti (ad esempio per il comportamento di ricerca) sono rappresentati da piccolissimi cerchi nella scena.
Dopo aver realizzato delle combinazioni di comportamenti interessanti, si è cercato di mettere in comunicazione l'applicazione con l'ambiente grafico 3D del progetto di simulazione di folle esistente, in modo da poter apprezzare l'applicazione in una veste grafica decisamente più avanzata.
Il sistema si presenta con una scena 3D costituita dall'immagine di un prato, cielo e montagne realizzata attraverso una cubemap. L'obiettivo principale del progetto di questo sistema era studiare il movimento di gruppi di bipedi costituiti da modelli animati di persone realizzati con 3ds Max ed esportati con Collada in formato .dae in modo da renderli compatibili con Horde3D. In tale ambiente erano presenti una serie di ostacoli rappresentati da diversi edifici reperiti attraverso GoogleSketchup ed anch'essi esportati in formato Collada (figura 4.2).

Figura 4.2: Visualizzazione dell'ambiente grafico 3D

I diversi elementi posti sulla scena rappresentano alcuni ostacoli da evitare e le possibili mete da raggiungere. Il problema fondamentale di tale sistema, affrontato in questa tesi, consiste nel fatto che i personaggi della simulazione riescono a

gestire le reciproche collisioni ma non hanno la capacità di percepire ed evitare un ostacolo. La possibilità di integrare questo sistema con l'applicativo realizzato per mezzo di Opensteer ha consentito di risolvere il problema descritto: infatti nella soluzione realizzata i pedoni percepiscono gli ostacoli e li evitano aggirandoli senza attraversarli.

La difficoltà tecnica principale riscontrata nella soluzione appena vista è dovuta al fatto che l'ambiente grafico, non essendo stato realizzato ad hoc, presenta diverse difficoltà di integrazione con le librerie Opensteer. I plugin realizzati in questa tesi presentano una serie di agenti collocati in basso nella scena con l'obiettivo principale di raggiungere un target posizionato in alto e con interposti due grossi ostacoli; nel percorso tra gli agenti e la meta sono posizionati altri ostacoli più piccoli. Gli agenti iniziano a muoversi in direzione dell'obiettivo ma, non appena incontrano i primi ostacoli, deviano la loro traiettoria in modo da non collidere con gli ostacoli e contemporaneamente devono evitarsi tra loro in modo da non compenetrarsi; infine, superati gli ostacoli centrali, devono dirigersi nuovamente verso il target e incanalarsi nel percorso obbligato creato dai due ostacoli più grossi, accodandosi per raggiungere il target. Ogni volta che un agente raggiunge l'obiettivo, esso viene riposizionato nel suo punto di partenza (in basso nella scena) come a simulare un flusso continuo che ripete lo stesso comportamento.

All'interno dell'applicazione è possibile interagire tramite tastiera per modificare l'angolazione e la posizione della telecamera e variare il punto di vista della scena; inoltre è possibile bloccare la scena, o farla ripartire dall'inizio. Infine, avendo creato diverse versioni di complessità incrementale, è possibile passare da un plugin all'altro. Per differenziare il flusso di agenti in tre target diversi è stata realizzata un'evoluzione di questo plugin, in modo da simulare il flusso di tre gruppi di agenti che si dirigono in diversi punti nella scena. E' stato anche creato un plugin che simula la pianificazione di semplici traiettorie all'interno della scena in modo tale che gli agenti siano indirizzati in target successivi (rappresentati da piccoli cerchi) e abbiano traiettorie indipendenti l'uno dall'altro, evitando la compenetrazione.

4.2 Sviluppo del sistema

I requisiti che dovranno essere soddisfatti sono i seguenti:

- Il sistema dovrà offrire i mezzi per evitare gli ostacoli posti sulla scena indipendentemente dalla loro posizione, numero o dimensione.
- Il sistema dovrà permettere agli agenti di evitarsi tra di loro anche in presenza di un elevato numero in una zona ristretta.
- Il sistema dovrà fornire agli agenti un comportamento intelligente (ad esempio se la strada per raggiungere una meta è troppo affollata potranno optare per lasciare libero il passaggio e riprovare ad accedere in seguito, oppure potranno cercare un'altra strada.)
- I requisiti richiesti nei punti precedenti dovranno poter essere soddisfatti anche contemporaneamente secondo un principio di assegnazione di priorità.
- L'integrazione della nuova simulazione non dovrà pesare sulle prestazioni della simulazione sviluppata originariamente presso il Laboratorio di Robotica del Dipartimento di Ingegneria dell'Informazione.

Per poter analizzare le librerie Opensteer e capirne il funzionamento e le possibilità è stato necessario installarle insieme alla demo esistente. Lo studio dei plugin forniti all'interno della demo è stato fondamentale per comprendere i diversi possibili utilizzi delle librerie. Dato che ogni forza di sterzo è rappresentata da un vettore in tre dimensioni, è stato importante analizzare a fondo la classe Vec3. Definito il comportamento combinato da realizzare, attraverso la fusione dei vettori di sterzo coinvolti, si è passati alla scrittura vera e propria dei plugin. Dopo aver progettato l'applicativo funzionante si è studiato un modo per mettere in comunicazione il plugin realizzato con il sistema di simulazione di folle.
Siccome l'ambiente utilizzato non è un semplice visualizzatore ma un sistema che gestisce le collisioni tra agenti, si è dapprima dovuto capire come evitare che influisse in modo imprevedibile sulle traiettorie previste dalla realizzazione fatta con Opensteer; successivamente si è elaborato un sistema che prelevasse

dall'applicazione di partenza le posizioni successive dei percorsi fatti dagli agenti, e li assegnasse ai personaggi tridimensionali all'interno del sistema 3D. Si è utilizzato un sistema intuitivo che è stato possibile applicare solo ad un numero limitato di personaggi. Nel sistema realizzato i modelli animati di gruppi di persone riescono a aggirare gli ostacoli.

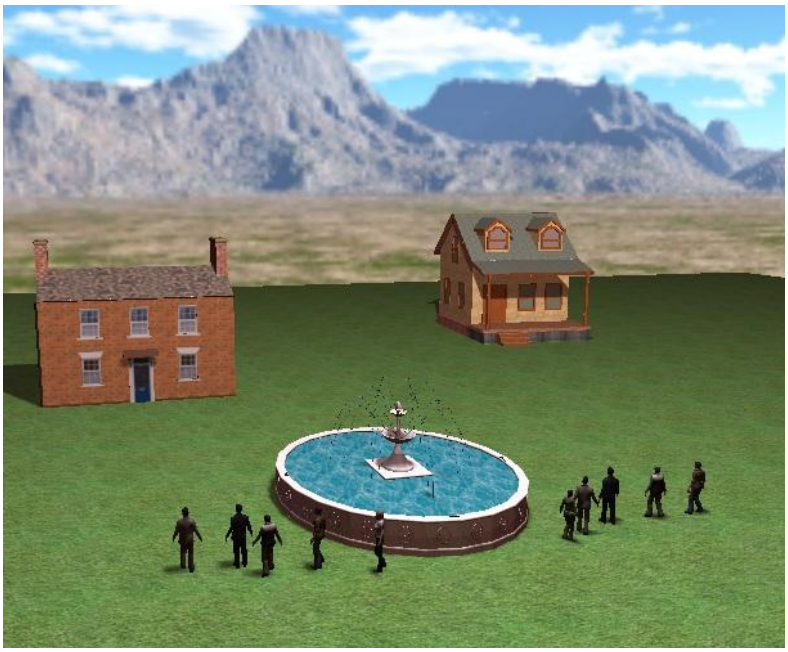

Figura 4.4: Immagine del lavoro realizzato integrato con l'ambiente 3D

Nella figura 4.4 si vede come i personaggi presi in esame riescano a cambiare la traiettoria di partenza in modo da evitare la fontana al centro della scena.
Il primo problema da affrontare è stato uniformare la posizione di partenza dei personaggi con la posizione e le dimensioni degli ostacoli da aggirare, in modo che le traiettorie realizzate dagli agenti del plugin con Opensteer fossero coerenti con le traiettorie seguite dai personaggi nell'ambiente 3D. Dato che la soluzione adottata sfrutta la scrittura su file, è sorto un ulteriore problema: per le traiettorie particolarmente lunghe ogni personaggio avrebbe prodotto un file di dimensioni considerevoli; inoltre il numero di personaggi deve essere stabilito a priori e non può essere aumentato o diminuito a piacere se non mettendo mani sul codice (sia di Opensteer che dell'ambiente 3D).

4.3 Struttura del codice

Il codice è strutturato attraverso un main che include le librerie Opensteer sfruttando Opensteerdemo e attraverso la classe Draw che viene utilizzata per la parte grafica. Il main si occupa di gestire l'inizializzazione della demo, della grafica e di eseguire il ciclo di elaborazione principale. Utilizzando delle chiamate a funzioni appartenenti alla classe Opensteer e OpensteerDemo, viene generato il loop dell'animazione e il susseguirsi dei diversi plugin presenti nell'applicativo.
I plugin sono organizzati in classi, all'interno delle quali vengono creati e usati gli agenti che ereditano dalla classe SimpleVehicle e AbstractVehicle. Viene inoltre creato l'oggetto PlugIn che eredita dalla medesima classe. Sono stati realizzati diversi plugin (figura 4.5) tra cui i più importanti sono Accodamento e Traiettoria.

4.3.1 Plugin Accodamento

Oltre alle variabili globali visibili su tutto il plugin, Accodamento è strutturato in due classi: la classe AccVehicle (che eredita da SimpleVehicle) e la classe AccPlugIn (che eredita da Plugin).

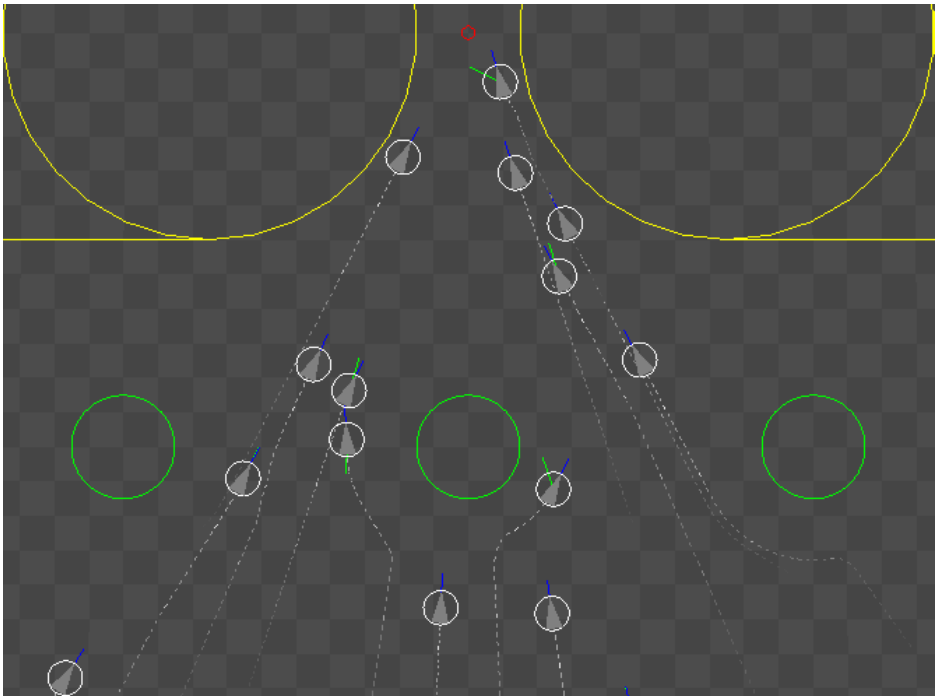

Figura 4.5: PlugIn Accodamento con un solo gruppo e una serie di ostacoli centrali

Più in dettaglio, la classe AccVehicle contiene:

- il costruttore che permette, per mezzo di un indice, la creazione di una lista di agenti;
- il reset che eredita il metodo reset presente nella classe Simplevehicle, inoltre imposta la massa iniziale, azzera la forza di sterzo iniziale, setta la spinta in avanti, la velocità iniziale, imposta la massima velocità (che servirà come parametro per troncare i valori troppo elevati), e imposta la posizione di partenza dei veicoli nella scena;
- il metodo update che viene aggiornato ad ogni passo della simulazione, e viene ripetuto individualmente per ogni agente. In questo metodo viene dapprima azzerata la forza di sterzo, poi si valuta se il traguardo è stato raggiunto; in caso affermativo l'agente viene riportato al punto di partenza in basso nella scena utilizzando il metodo reset per realizzare l'idea di un flusso continuo di agenti oppure, in alternativa, viene valutata la necessità di evitare eventuali ostacoli presenti nel percorso. Se questo controllo percepisce qualche ostacolo presente sul percorso nella direzione in cui si sta muovendo il veicolo, viene inserita nella variabile "obstacle" il valore dello sterzo necessario per evitarlo. A questo punto viene valutata questa variabile per attribuirne il valore alla forza di sterzo "steeringforce". In seguito viene richiamata la funzione annotateavoidVehicle che valuta la necessità di evitare veicoli che possono essere in rotta di collisione col veicolo considerato, e in caso affermativo restituisce il valore dello sterzo correttivo nella variabile "avoid"; infine valuta se i due parametri "obstacle" e "avoid" contengono valori diversi da zero: in tal caso viene fatta la loro somma considerando un fattore di ponderazione per l'ostacolo (pari a 3), ottenendo il valore della forza di sterzo. Se uno dei due parametri considerati risultasse nullo, allora la forza di sterzo sarebbe semplicemente la somma dei due (che in questo caso equivarrebbe a considerare esclusivamente la variabile non nulla). Infine se entrambi i valori sono pari a zero, ovvero non ci sono né ostacoli né altri veicoli da evitare, all'agente può essere applicata la forza di sterzo relativa alla ricerca dell'obbiettivo.

```
// aggiornamento simulazione
void update (const float currentTime, const float elapsedTime)
{
            this->steeringForce = Vec3::zero;
            // ha raggiunto l'obbiettivo!
            if(Vec3::distance(this->position(), ACendPosition) < ACtouchDistance ||
            this->position().x > ACcutX)
            {
                    this->reset();
            }
            Vec3 obstacle, avoid; // forza per evitare l'ostacolo
            float oTime = 0.3f;
            // controlla l'ostacolo 1
            obstacle = steerToAvoidObstacle(oTime, ob1);
            // controlla l'ostacolo 2
            if(obstacle == Vec3::zero)
                    obstacle = steerToAvoidObstacle(oTime, ob2);
            // controlla l'ostacolo 3
            if(obstacle == Vec3::zero)
                    obstacle = steerToAvoidObstacle(oTime, ob3);
            // controlla l'ostacolo 4
            if(obstacle == Vec3::zero)
                    obstacle = steerToAvoidObstacle(oTime, ob4);
            // evitamento ostacoli piccoli posizionati immezzo alla scena
            if(obstacle == Vec3::zero)
                    obstacle = steerToAvoidObstacle(oTime, miniob1);
            if(obstacle == Vec3::zero)
                    obstacle = steerToAvoidObstacle(oTime, miniob2);
            if(obstacle == Vec3::zero)
                    obstacle = steerToAvoidObstacle(oTime, miniob3);
            // imposta la forza di guida per evitare l'ostacolo
            // se uno dei due è stato individuato dal veicolo
            if(obstacle != Vec3::zero)
            {
                    this->steeringForce = obstacle;
            }
            avoid = annotateAvoidVehicle(); // evitamento altri veicoli
            // se è necessario sia evitare ostacolo che altri veicoli
            // viene calcolata la somma ponderata delle due forze di sterzo
            if(avoid != Vec3::zero && obstacle != Vec3::zero)
            {
                    // somma ponderata
                    this->steeringForce += avoid + obstacle*3;
            }
            // altrimenti somma alla forza di guida quella necessaria
            // (o l'evita vicino o l'evita ostacolo)
            // ovviamente in questo caso uno dei due sarà =0
            // quindi non influirà sulla somma
            else
            {
                    this->steeringForce += avoid + obstacle;
            }
            // se non è necessario né evitare vicini né ostacoli
            if(this->steeringForce == Vec3::zero)
            {
                    // si imposta la forza di guida per cercare l'obiettivo!
                    this->steeringForce = steerForSeek(ACendPosition);
            }
            // applica forza di guida
            applySteeringForce(this->steeringForce,elapsedTime);
            recordTrailVertex (currentTime, position());
}
```

- il metodo draw disegna il veicolo all'interno della scena rappresentandolo come un cerchio sul piano XZ del colore desiderato (in questo caso grigio) e rappresenta la velocità del veicolo con un segmento blu mentre la forza di sterzo è raffigurata con segmenti di colore verde applicati sul veicolo.
- annotateAvoidVehicle permette di determinare la forza di guida necessaria ad evitare gli altri veicoli oppure restituisce un vettore zero (se non c'è pericolo di collisioni). Per ogni veicolo presente nella scena viene impostato il valore "distance" che rappresenta la distanza rispetto all'agente preso in considerazione; successivamente questo valore viene confrontato con il valore del raggio del veicolo (eventualmente moltiplicato per un fattore che stabilisce la distanza che si desidera mantenere tra i veicoli) e, se la distanza è minore di quella minima desiderata, viene calcolata la forza di sterzo necessaria ad allontanare i due veicoli presi in considerazione.

Un'altra classe molto importante è AccPlugIn che è così strutturata:

- il metodo open crea inizialmente, attraverso un ciclo for, una lista di "i" agenti e quindi, attraverso le funzioni ereditate da OpenSteerDemo, inizializza la telecamera, la sua posizione, e la modalità di visualizzazione.
- il metodo update serve per aggiornare lo stato di tutti gli agenti ad ogni passo della simulazione; si procede con un ciclo for che, per ogni agente, invoca il metodo update (appartenente alla classe AccVehicle).
- il redraw, attraverso un ciclo for, aggiorna l'immagine di tutti gli agenti nella scena. Inoltre aggiorna la telecamera, disegna l'obbiettivo che gli agenti devono raggiungere, attraverso un piccolo cerchio rosso; si incarica inoltre di disegnare gli ostacoli nella scena (come grossi cerchi gialli in alto o cerchi verdi di medie dimensioni posizionati al centro) ed inoltre disegna il piano.
- il metodo close elimina uno per uno i puntatori della lista di veicoli cancellandone gli elementi.
- il metodo reset richiama per ogni agente il metodo reset della classe AccVehicle, in questo modo gli agenti vengono inizializzati.

Infine all'interno della classe AccPlugIn è presente un vettore di puntatori a veicoli, che servirà per accogliere (attraverso i puntatori) tutti gli agenti in fase di creazione, utilizzo e infine cancellazione.
Il plugin realizzato inizialmente e descritto in precedenza è stato ulteriormente arricchito in modo da creare delle varianti più interessanti e complesse: è stato organizzato il codice in modo da creare due o tre gruppi di agenti (rispettivamente figura 4.6 e 4.7) ai quali è assegnata una destinazione diversa. I gruppi di agenti in questione sono rappresentati da un triangolino di colore diverso (bianco, grigio, e nero). In fase di collaudo è stato interessante vedere come modificando la meta finale, in modo da intrecciare il percorso dei gruppi, si creava un effetto di stallo iniziale che comunque si risolveva ottenendo che ogni agente raggiungesse la destinazione richiesta.

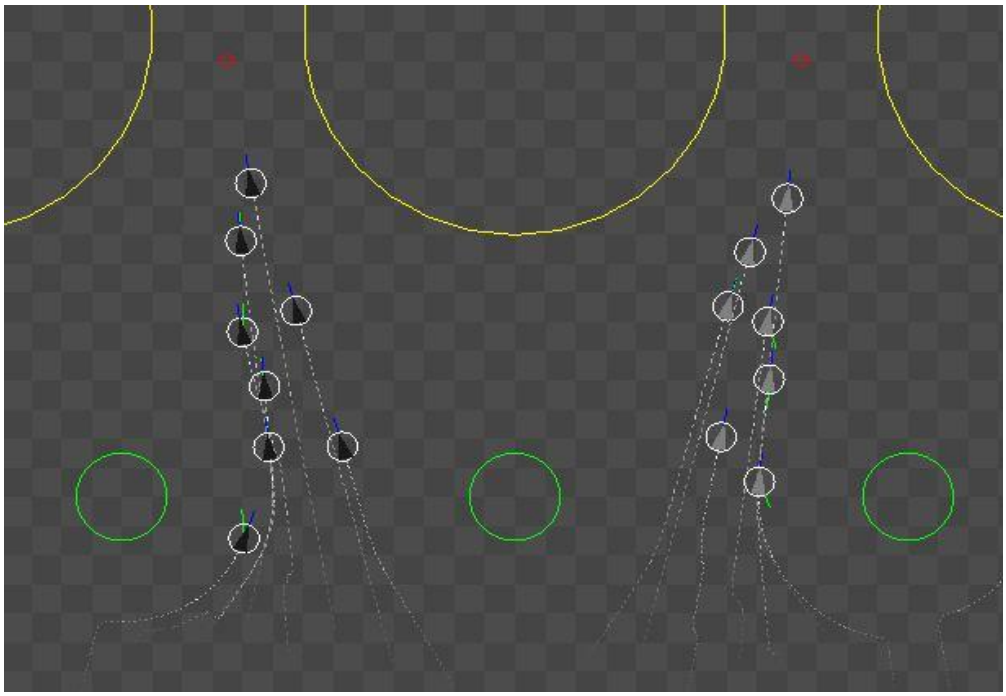

Figura 4.6: Plugin con due gruppi indirizzati a target distinti

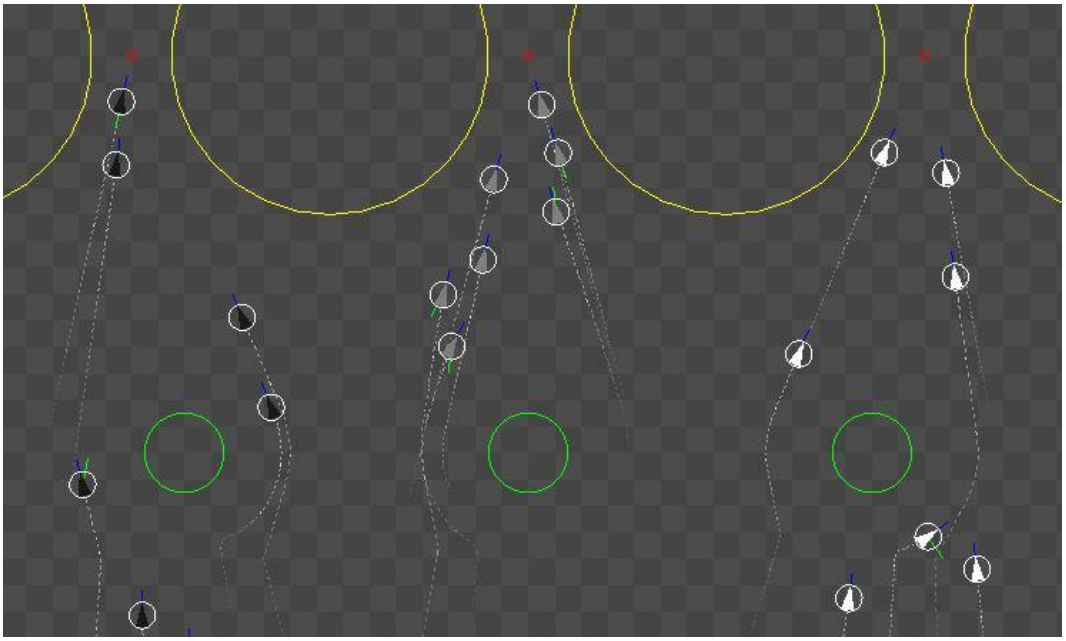

Figura 4.7: Plugin Accodamento con tre gruppi

4.3.2 Plugin Traiettorie

Anche il plugin Traiettorie è strutturato in due classi, in questo caso abbiamo TRVehicle (che eredita da SimpleVehicle) e la classe TRPlugIn (che eredita sempre da plugin).
Vediamo nel dettaglio le classi considerate. La classe TRVehicle è strutturata in modo molto simile a AccVehicle, ma implementata in modo diverso:

- il costruttore è analogo a quello già presentato nella classe AccVehicle.
- il reset innanzitutto imposta la massa, la forza di sterzo e la velocità iniziali pari a zero, poi vengono acquisiti su un array le posizioni che costituiranno i target nell'ordine desiderato per la traiettoria da seguire. Infine vengono impostate le posizioni iniziali degli agenti.
- il metodo update inizialmente imposta a zero la forza di sterzo. Poi uno switch (in base al valore relativo all'indice "indextarget") definisce a quale target si è arrivati fino a quel momento e stabilisce se l'agente ha già raggiunto il target successivo. In caso affermativo viene semplicemente incrementato il valore dell'indice indextarget, così l'agente si dirigerà verso il punto successivo previsto per la traiettoria. Dato che questa

operazione viene ripetuta indipendentemente per ogni agente, non ci sarà il rischio che quando il primo agente raggiungerà il primo target e si dirigerà verso il secondo, possa cambiare questo parametro anche per gli altri; ma ognuno procederà in maniera indipendente. Anche in questo caso devono essere considerate le potenziali reciproche collisioni, quindi viene richiamato il metodo annotateAvoidVehicle per valutarle. Il risultato (che potrà essere il vettore di sterzo o un vettore nullo) verrà inserito nella variabile avoid e poi valutato per essere applicato o meno al veicolo.

```
// aggiornamento simulazione
void update (const float currentTime, const float elapsedTime)
{
            this->steeringForce = Vec3::zero;
            // verifica se ha raggiunto l'obbiettivo!
            switch (indexcut)
            {
            case(0):
            if(Vec3::distance(this->position(), target[indextarget]) < TRtouchDistance
            || this->position().x > cut[indexcut])
            {
                    indextarget++;
                    indexcut++;
            }
            break;
            case(1):
            if(Vec3::distance(this->position(), target[indextarget]) < TRtouchDistance
            || this->position().x < cut[indexcut])
            {
                    indextarget++;
                    indexcut++;
            }
            break;
            case(2):
            if(Vec3::distance(this->position(), target[indextarget]) < TRtouchDistance
            || this->position().x > cut[indexcut])
            {
                    indextarget++;
                    indexcut++;
            }
            break;
            case(3):
            if(Vec3::distance(this->position(), target[indextarget]) < TRtouchDistance
            || this->position().x < cut[indexcut])
            {
                    indextarget=0;
                    indexcut=0;
            }
            break;
            }
            // end switch
            Vec3 avoid; // forza per evitare veicoli vicini
            Vec3 obstacle; // forza per evitare l'ostacolo
            float oTime = 0.3f;
            obstacle = steerToAvoidObstacle(oTime, ob1); // controlla l'ostacolo 1
            // imposta la forza di guida per evitare l'ostacolo
            // se è stato individuato dal veicolo
            if(obstacle != Vec3::zero)
            {
                    this->steeringForce = obstacle;
            }
            avoid = annotateAvoidVehicle(); // evitamento altri veicoli
```

```
            // se è necessario sia evitare ostacolo che altri veicoli
            if(avoid != Vec3::zero && obstacle != Vec3::zero)
            {
                    // somma ponderata
                    this->steeringForce += avoid + obstacle*3;
            }
            // altrimenti somma alla forza di guida l'evitamento necessario
            // (vicino o ostacolo)
            // in questo caso uno dei due sarà =0 quindi non influirà sulla somma
            else
            {
                    this->steeringForce += avoid + obstacle;
            }
            // se non è necessario né evitare vicini né ostacoli
            if(this->steeringForce == Vec3::zero)
            {
                    // si imposta la forza di guida per cercare l'obiettivo!
                    // l'obiettivo è di volta in volta diverso in base a quale punto
                    // del percorso è stato raggiunto
                    this->steeringForce = steerForSeek(target[indextarget]);
            }
            // applica forza di guida
            applySteeringForce(this->steeringForce,elapsedTime);
            recordTrailVertex (currentTime, position());
}
```

- il metodo draw è analogo al metodo draw visto per la classe AccVehicle, in quanto serve a disegnare il veicolo sulla scena e i segmenti che rappresentano la velocità e la forza di sterzo.
- annotateAvoidVehicle è anch'essa analoga all'omonima presente nella classe AccVehicle.

Le caratteristiche della classe TRPlugIn sono le seguenti:

- i metodi open e update sono identici a quelli presenti nella classe AccPlugIn e anche qui servono per creare la lista di veicoli, inizializzare la telecamera e aggiornare tutti gli agenti ad ogni passo dell'animazione.
- il redraw è idealmente analogo al metodo omonimo presente nella classe AccPlugIn, ma sostanzialmente diverso in quanto gli oggetti presenti nella scena sono differenti. Questo metodo, attraverso un ciclo for, aggiorna l'immagine di tutti gli agenti nella scena; inoltre aggiorna la telecamera, disegna tutti i target successivi che costituiscono la traiettoria da seguire ed infine disegna il piano.
- anche i metodi close e reset sono analoghi ai loro omonimi presenti nella classe AccPlugIn e serviranno ad eliminare i puntatori della lista di veicoli

cancellandone gli elementi e a resettare gli agenti in fase di inizializzazione.

Anche con il plugin relativo alla traiettoria degli agenti sono state fatte diverse prove, tra cui obbligare i veicoli a seguire una traiettoria a forma di rombo o quadrato, oppure inserire ostacoli nelle zone tra un target e il successivo in modo da costringere i veicoli a deviare per poi ritornare sulla propria rotta come si vede nelle figure 4.8 e 4.9.

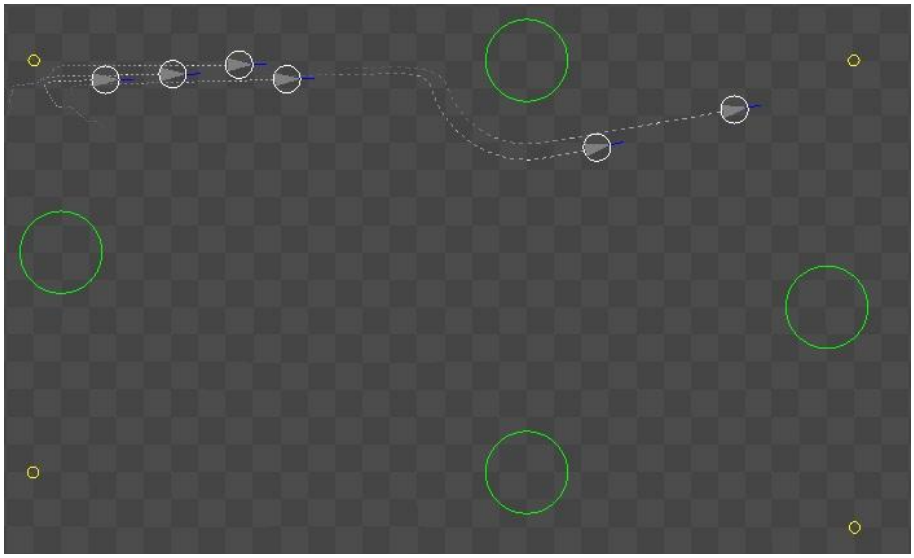

Figura 4.8: Plugin Traiettoria rettangolare con ostacoli

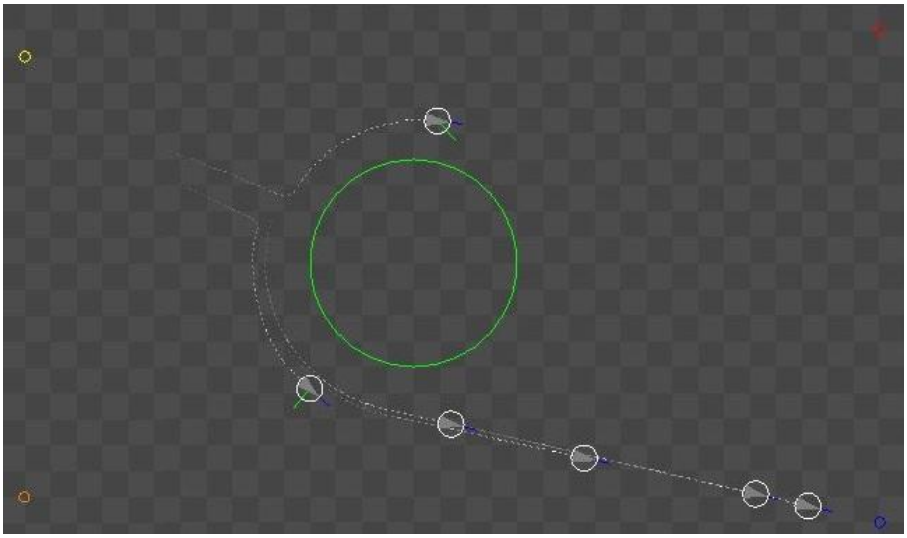

Figura 4.9: Plugin Traiettoria con target in diverso ordine e ostacolo centrale

4.4 Integrazione con il visualizzatore grafico

Data la necessità di presentare le scene con un aspetto più gradevole e realistico, si è cercato un modo per integrare questi plugin con un progetto grafico esistente finalizzato alla simulazione di folle. Si è dovuto studiare il codice di questo progetto per capire come venivano gestiti i movimenti degli modelli umani e come assegnare la posizione relativa agli agenti del plugin realizzato attraverso le librerie di Opensteer.

Si è studiato un modo per recuperate le posizioni successivamente occupate da ogni agente su Opensteer e applicarle forzatamente ai modelli umani del gruppo. A questo proposito si è creata una scrittura su file dal plugin di interesse prelevando le posizioni degli agenti ad ogni fotogramma. Quindi si è sfruttata la funzione redraw del plugin in cui, tramite un ciclo for, vengono aggiornati gli agenti ad ogni passo di visualizzazione. In tal modo sono stati creati tanti file quanti agenti si vuole animare e sono stati aperti in modalità append ovvero lasciando intatto quello che è presente nel file e scrivendo alla fine. In questo modo si evita che, ad ogni iterazione del ciclo for, venga cancellato e riscritto un nuovo file ma ne viene creato uno solo contenente tutte le posizioni successive dell'agente. Per evitare di mischiare o confondere le traiettorie di un agente con quelle degli altri si è tenuta traccia dell'iteratore relativo al for e, grazie ad uno switch, si è potuto differenziare le traiettorie scrivendo nel "file0.txt" le posizioni successive del primo agente, nel "file1.txt" quelle relative al secondo e così via.

Per recuperare e inserire nel file le posizioni degli agenti si è sfruttato il metodo position ereditato da AbstractVehicle. In questo modo è stato possibile recuperare la posizione relativa alle coordinate X, Y e Z. Ma dato che il nostro veicolo (e anche i modelli umani presenti nel visualizzatore) si muoveranno su piano (non voleranno in aria né sprofonderanno nel terreno) in realtà abbiamo bisogno solo delle posizioni relative a X e Z (la coordinata su Y resta costantemente pari a zero).

Un altro aspetto importante ai fini pratici è come salvare su file le coordinate acquisite. Dato che il modo migliore per recuperarle con l'applicativo di visualizzazione è utilizzare il metodo "getline" (che recupera riga per riga il

contenuto del file) allora si è deciso di acquisire le coordinate in righe alternando la coordinata su X a quella su Z.

Ovviamente si è dovuto pensare a un modo efficace per recuperarli e organizzarli nell'altro applicativo. A questo proposito si è creato un gruppo di 2 vettori per ogni file da gestire (nel nostro caso avremo 10 vettori) e una funzione chiamata LeggiFile. Tale funzione si preoccupa di aprire ognuno dei 5 file esplorando tutte le righe (attraverso un while e la funzione good).

In questo modo all'interno del ciclo viene prelevata una riga attraverso la funzione getline, trasformata la stringa in un float attraverso "stringstream" e poi attribuita alla prima posizione dell'array relativo alla coordinata X del file0. All'interno dello stesso ciclo di iterazione viene estratta una nuova riga, sempre con la funzione getline, questa volta relativa però alla prima coordinata su Z del file0. Con lo stesso procedimento di estrazione dal file e trasformazione in float, il valore sarà poi attribuito alla prima posizione del vettore relativo alla coordinata Z del primo file. Solo a questo punto è opportuno incrementare l'iteratore relativo all'indice dell'array. In questo modo ad ogni iterazione del while vengono estratte due righe dal file che sono le due coordinate relative alla stessa posizione. Sono quindi inserite nella stessa posizione dei due array rispettivamente X e Z.

Questa procedura viene ripetuta per ogni file (nel nostro caso 5 volte). Dopo aver effettuato la lettura da file bisogna assegnare le posizioni successive ai modelli umani. Dato che il sistema tridimensionale è costituito da numerosi gruppi si è tentato in un primo momento di utilizzare un solo gruppo bloccando gli altri e assegnando le posizioni contenute nell'array. A tale scopo si è tentato di modificare la classe group; attraverso uno switch si è controllato l'identificatore del modello umano e assegnata la posizione iniziale o incrementale rispettivamente nella funzione init e update. Questa soluzione, in verità, generava degli errori sconosciuti e i modelli umani sparivano dalla scena. Si è dovuto quindi cercare un modo alternativo di gestire le posizioni successive. Dato che il sistema tridimensionale supportava più di un gruppo si è pensato quindi di utilizzarne 5 composti ognuno da un solo elemento. A questo punto si è assegnata come destinazione la serie di punti successivi caricati dal file all'array, distinguendo i modelli umani attraverso il gruppo di appartenenza. A tale scopo si è dapprima disabilitato il controllo di "collision detection" e le forze di repulsione

esistenti su questo sistema, in modo che non influenzassero le traiettorie definite dal plugin di Opensteer. Successivamente si è aggiunto un parametro alla funzione update della classe group in modo da tenere traccia del numero identificativo del gruppo (chiamato igroup) gestito nel for del metodo "mainLoop" della classe "application". In questo modo, nella classe update della classe group, si può stabilire l'azione: se il personaggio non ha ancora raggiunto la destinazione assegnata, deve camminare per raggiungerla altrimenti, se ha raggiunto la destinazione desiderata, gli verrà assegnata la destinazione successiva dall'array delle posizioni.

```
void Group::update( float fps, bool _bubbleMode, bool _speedyMode , int igroup)
{
	int counter = 0;
	//Inizialise the new barycenter cohordinates
	float newBx = 0;
	float newBz = 0;
	float pdx;
	float pdz;
	for( unsigned int i = 0; i < _people.size(); ++i )
	{
		Person &p = _people[i];
		float px = p.getPx();
		float pz = p.getPz();
		float ox = p.getOx();
		float oz = p.getOz();
		float dx = p.getDx();
		float dz = p.getDz();
		// Reset force
		// *** FORZA 0 ***
		float fx = 0;
		float fz = 0;
		// Calculate distance to destination
		float dist = sqrtf( (dx - px)*(dx - px) + (dz - pz)*(dz - pz) );

		// If destination not reached walk to destination
		if( dist > 0.2f ) //era 3.0f
		{
			// Calculate normalized attraction force to destination
			float afx = (dx - px) / dist;
			float afz = (dz - pz) / dist;
			//Forza attrattiva verso destinazione
			// *** FORZA 1 ***
			fx += afx * 0.035f;
			fz += afz * 0.035f;
		}
		else
		{
		// Choose new destination
			switch (igroup)
			{
			case(0):
			// boids numero 0
				{
					pdx=posizionex0[pos0];
					pdz=posizionez0[pos0];
					p.setDestination(pdx, pdz);
					pos0++;
				}
			break;
			case(1):
```

```
        // boids numero 1
                {
                ...
                }
        break;
        ... // stessa cosa ripetuta per ogni omino (nel nostro caso 5)
        } // end switch
} // end else
// Make movement framerate independent
int speed = _speedyMode? 160 : 30;
fx *= (speed / fps);
fz *= (speed / fps);
p.setForce(fx,fz);
float vel = sqrtf( fx * fx + fz * fz );
// Set new position
px += fx;
pz += fz;
p.setPosition(px,pz);
// Update bx and bz values (I want to compute the mean value
// of the positions of all members of the group)
newBx += px;
newBz += pz;
// Calculate orientation
ox = (ox + fx) / 2;
oz = (oz + fz) / 2;
p.setOrientation(ox,oz);

...             // il resto della funzione si può trovare
                // nella documentazione allegata alla tesi
}
```

Per assegnare la destinazione successiva al modello umano dall'array delle posizioni si è usata la funzione setDestination della classe person. Inoltre si è modificata la funzione chooseDestination che inizialmente prendeva in ingresso un parametro (la persona) e restituiva una destinazione casuale. Dopo le modifiche fatte, la funzione prende in ingresso la persona e la posizione (attraverso coordinate x e z) e assegna la coordinata alla persona considerata attraverso la funzione setDestination.
La fedeltà alla traiettoria definita da Opensteer, dipende dal numero di posizioni che si acquisiscono attraverso i file.

4.4.1 Problemi di integrazione con l'applicazione di visualizzazione 3D

L'integrazione dei due sistemi ha presentato diverse difficoltà, sia riguardo alla dimensione e posizione degli oggetti che in relazione alle dimensioni dei file generati.

Relativamente alle dimensioni si è rilevato che le unità di misura non corrispondono: in Opensteer le dimensioni dei veicoli e gli spazi percorsi sono molto più piccoli delle dimensioni dei modelli umani e degli spazi percorsi da questi nel visualizzatore 3D. Per risolvere questo problema si è dovuto procedere per tentativi al fine di determinare il fattore di scala e modificare i parametri relativi al plugin basato su Opensteer. Le modifiche apportate sono riportate nella tabella seguente.

	default	per il visualizzatore
coordinata X di partenza	-10	-25
coordinata X di arrivo	11	100
distanza percorsa	21	125
raggio ostacolo	3	18
distanza reciproca tra agenti	3	10

Il secondo problema dipende dal fatto che, rilevando tutte le posizioni degli agenti, i file generati assumevano dimensioni considerevoli. Per risolvere tale problema si è pensato di ridurre il numero di posizioni acquisite da file prendendone ad esempio una ogni 10. Successivamente si è analizzato l'andamento della fedeltà delle traiettorie in 3D in relazione al fattore di approssimazione scelto. C'era il timore che prendendo un numero sostanzialmente inferiore di posizioni dal plugin di Opensteer, i modelli umani avrebbero seguito meno fedelmente la traiettoria desiderata. Questo è vero solo in parte; si è scoperto infatti, facendo un congruo numero di test, che il numero di punti può essere significativamente ridotto senza modificare sensibilmente le traiettorie.

4.4.2 Analisi delle prestazioni

Per valutare l'affidabilità delle traiettorie acquisite da Opensteer al variare delle approssimazioni fatte, si è aggiunto nel codice del plugin la scrittura di file che

preleva simultaneamente i dati (relativi a un agente) con un'approssimazione di 10, 50, 100, 1000 e 2000. Si è deciso di acquisire i dati simultaneamente perché il plugin di Opensteer essendo basato su un prototipo di intelligenza artificiale non riproduce in modo statico i percorsi degli agenti, ma li elabora in tempo reale. In questo modo si ottiene che, facendo partire due volte lo stesso plugin, il percorso degli agenti è ogni volta diverso sebbene basato, ovviamente, sugli stessi comportamenti. Infatti, per poter ottenere risultati dei test significativi, si devono paragonare dati acquisiti nella stessa simulazione. Il grafico mostrato in figura 4.10 riporta le diverse traiettorie, ottenute nel visualizzatore 3D, relative alla stessa simulazione ma ad approssimazioni diverse.

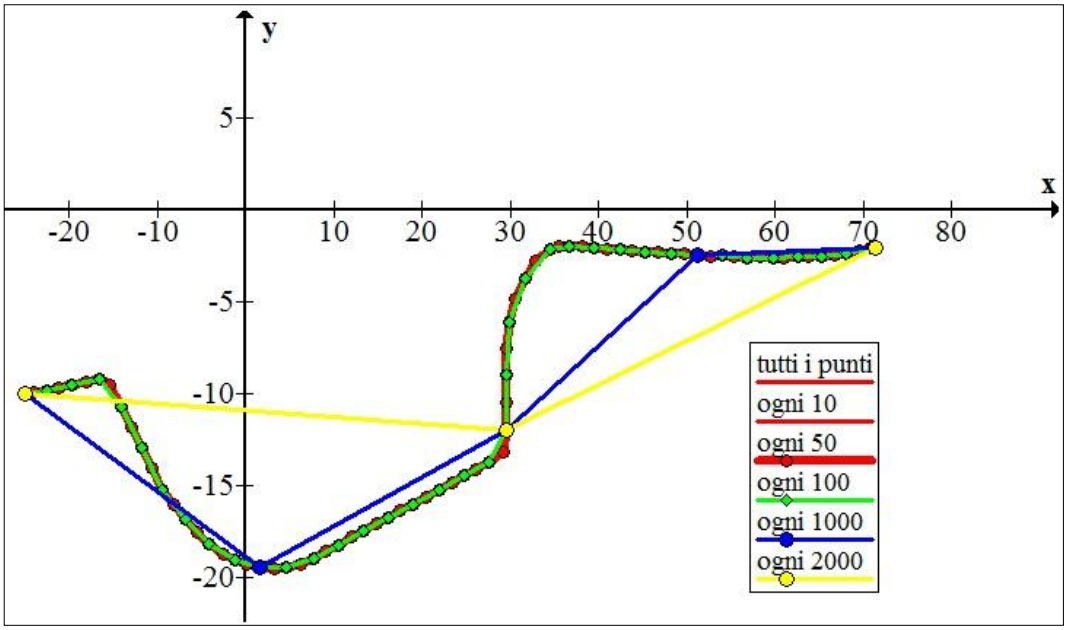

Figura 4.10: Grafico relativo alla valutazione delle traiettorie con approssimazione sui campioni

Il risultato sottolinea che, per file nei quali si prende un campione ogni 10, ogni 50 e ogni 100, la traiettoria seguita è abbastanza fedele all'originale, mentre per un'approssimazione superiore (uno ogni 1000 o più) la traiettoria si discosta completamente da quella di partenza. In particolare come si vede dal grafico, le traiettorie blu e gialla comporterebbero errori non accettabili in quanto i modelli umani non riuscirebbero ad evitare l'ostacolo ma cercherebbero di attraversarlo a causa del numero insufficiente di punti acquisiti per ricostruire il percorso. La traiettoria verde, che corrisponde all'acquisizione di un punto ogni 100, è

abbastanza fedele (ci sono solo dei piccoli tratti in cui il tragitto si discosta leggermente dall'originale senza compromettere il risultato finale); inoltre tale approssimazione permette di diminuire sensibilmente le dimensioni dei file. Dai risultati ottenuti, per un'animazione di circa 50 secondi (il tempo necessario ai modelli umani di aggirare quasi completamente l'ostacolo fontana) il numero di punti da assegnare per seguire la traiettoria sono relativamente pochi.
In particolare per l'animazione presa in esame le traiettorie risultate dal plugin di Opensteer comprendevano un numero di circa 8200 punti. Si è provato quindi a ridurli considerevolmente prendendo i campioni una volta ogni 10 (ottenendo quindi circa 820 posizioni), una volta ogni 50 (ottenendone 160), e poi ogni 100, 1000 e 2000. In questo modo le dimensioni dei file diminuiscono considerevolmente. Come si vede dal grafico in figura 4.11 le dimensioni del file si abbassano drasticamente al diminuire della quantità di campioni acquisiti.

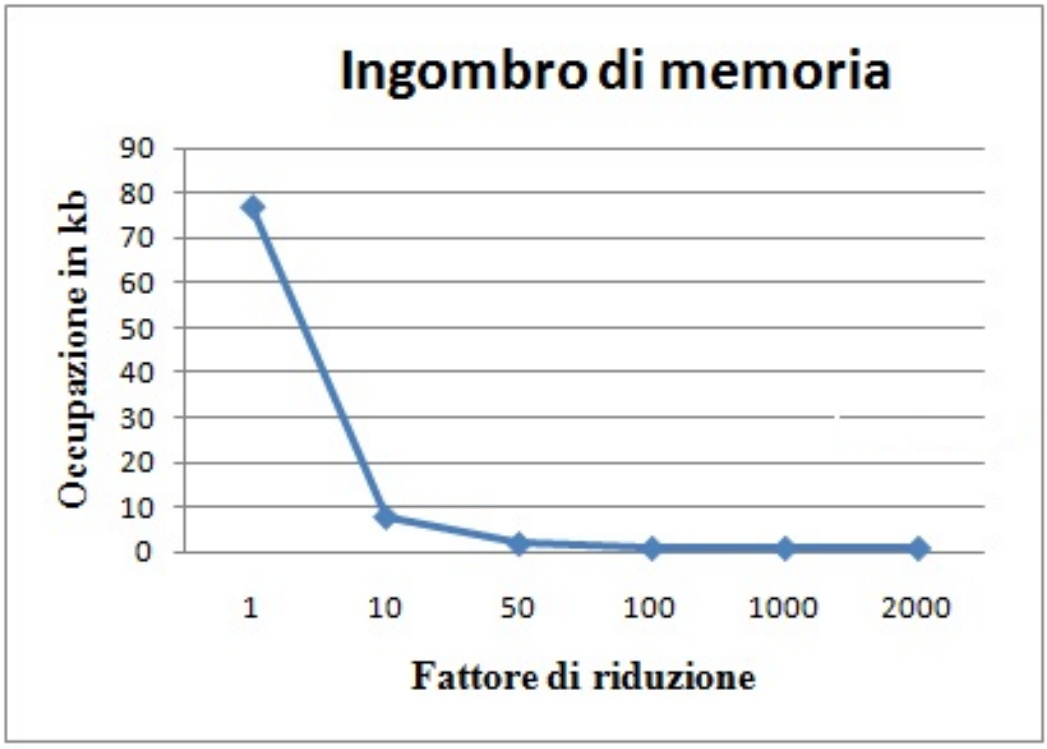

Figura 4.11: Grafico relativo all'ingombro di memoria in funzione dell'approssimazione

Anche se la dimensione di 80 kilobyte può sembrare irrilevante, essa si riferisce a un'animazione della durata di 50 secondi e relativa a un solo agente. Quindi la riduzione delle dimensioni dei file è fondamentale in considerazione del fatto che

tali dimensioni aumenterebbero esponenzialmente per animazioni più lunghe e con un numero elevato di agenti.

1 agente: 80 kb → 50 secondi
480 kb → 5 minuti (300 secondi)

10 agenti: 800 kb → 50 secondi
4800 kb ≈ 5 MB → 5 minuti

100 agenti: 8000 kb → 50 secondi
48000 kb ≈ 50 MB → 5 minuti

Da queste considerazioni si evince che la scelta ottimale è quella di prendere un campione ogni 100 perché questa scelta non compromette eccessivamente la fedeltà della traiettoria, riducendo notevolmente lo spazio occupato dal file e il peso delle strutture dati utilizzate nell'applicativo. In questo caso, utilizzando l'approssimazione di 1 a 100 campioni, l'ingombro in memoria sarebbe significativamente minore:

1 agente: 1 kb → 50 secondi
6 kb → 5 minuti (300 secondi)

10 agenti: 10 kb → 50 secondi
60 kb → 5 minuti

100 agenti: 100 kb → 50 secondi
600 kb → 5 minuti

L'importanza delle considerazioni suesposte non riguardano solo l'occupazione di memoria su disco rigido, ma rivestono notevole importanza parlando di memoria RAM. Se un programma richiede una memoria eccessiva la sua corretta elaborazione potrebbe essere compromessa oppure potrebbe richiedere tempi molto elevati. Sicuramente un file più pesante corrisponde a un numero più

elevato di dati che dovranno essere salvati in un'apposita struttura all'interno del programma. Questo comporta la necessità di avere strutture dati più efficienti. Ovviamente la scelta definitiva dipenderà da quanto è importante la precisione della traiettoria ed in alcuni casi si può decidere di accettare una maggiore pesantezza nella quantità di dati al fine di ottenere una traiettoria più precisa.

Capitolo 5
Conclusioni

5.1 Conclusioni e possibili sviluppi futuri

Questa tesi ha avuto come scopo la realizzazione di comportamenti di guida combinati per veicoli autonomi. Si è riusciti a coordinare in modo autonomo il movimento di agenti all'interno di una scena. Il sistema presenta agenti capaci di individuare ostacoli posti nella scena ed evitarli. Gli agenti inoltre riescono ad evitarsi reciprocamente, in modo da non collidere gli uni con gli altri, anche in presenza di tanti agenti in una zona ristretta. Sono state realizzate delle semplici traiettorie composte da uno o più target da raggiungere. Si è riusciti a soddisfare le richieste anche contemporaneamente secondo un principio di somma ponderata. Inoltre si è riusciti ad integrare il sistema con un simulatore grafico 3D in modo da rendere il risultato finale visivamente gradevole.
Il lavoro svolto si presta ad alcuni possibili miglioramenti e utilizzi concreti della presente realizzazione:

- Il progetto può essere adottato come software di base per coordinare il movimento di piccoli robot in uno spazio limitato. Il sistema prevede infatti il raggiungimento di posizioni determinate attraverso coordinate spaziali. È previsto inoltre che gli agenti evitino gli ostacoli e anche gli

altri agenti nelle vicinanze e queste sono le condizioni di base per il movimento in un ambiente (ad esempio una stanza) evitando le collisioni con oggetti o altri robot presenti. Il funzionamento potrebbe essere implementato attraverso l'uso di laser che consentono l'acquisizione a priori delle posizioni degli oggetti nella scena. In tale scenario il sistema esegue gli algoritmi che determinano dinamicamente il comportamento degli agenti (robot) in base a ciò che hanno intorno e invia i dati ai robot attraverso una struttura client-server in modo da indirizzarli nel movimento.

- Il progetto può essere esteso sviluppando nuovi comportamenti grazie a Opensteer in modo da riuscire a ricreare situazioni diverse. Risulterebbe stimolante creare categorie di agenti che reagiscono in base ad eventi innescati dal comportamento degli altri agenti. In questo modo si potrebbe simulare ad esempio un gregge di pecore al pascolo che scappa dopo l'arrivo di un branco di lupi, oppure la difesa di una truppa in seguito all'attacco dell'esercito nemico.

- L'integrazione con il visualizzatore 3D ha presentato notevoli difficoltà perché l'ambiente grafico non era stato creato ad hoc e pertanto è stato abbastanza complesso far interagire i due sistemi. Recentemente è emersa l'esistenza di un ambiente grafico Panda3D rilasciato con una versione delle librerie Opensteer in linguaggio Python chiamate pyOpenSteer. Con l'utilizzo di tale ambiente grafico si sarebbe potuto ottenere un risultato più interessante, con l'unico sforzo di tradurre gli algoritmi sviluppati dal linguaggio C++ al Python. Utilizzando questo strumento si sarebbe inoltre presentata la possibilità di proporre l'ambiente grafico non solo come un insieme di modelli animati di persone ma di automobili nel traffico, di greggi o mandrie al pascolo, o ancora branchi di pesci, o stormi di uccelli.
  È da sottolineare che questa attuale versione delle librerie pyOpenSteer permette anche l'integrazione con Blender, un software open source per la modellazione di contenuti 3D.

Bibliografia

(1). http://it.wikipedia.org/wiki/Metal_Gear_Solid

(2). http://www.xnaitalia.com/

(3). http://www.winphoneitalia.com/articoli/windows-phone-7/xna/steering-behaviors.aspx

(4). http://it.wikipedia.org/wiki/Realt%C3%A0_virtuale

(5). Jacques Ferber, Multi-Agent Systems: An Introduction to Artificial Intelligence, Addison-Wesley, 1999.

(6). Gerhard Weiss, a cura di, Multiagent Systems, A Modern Approach to Distributed Artificial Intelligence, MIT Press, 1999

(7). Wiener, Norbert (1948) Cybernetics, or control and communication in the animal and the machine. Cambridge, Massachusetts: The Technology Press; New York: John Wiley & Sons, Inc.

(8). Reynolds C. W. Flocks, herds and schools: a distributed behavioral model. 1987, SIGGRAPH

(9). Still, G Keith (1994) “Simulating Egress using Virtual Reality - a Perspective View of Simulation and Design”, IMAS Fire Safety on Ships.

(10). Sims, Karl (1994) “Evolving Virtual Creatures”, Proceedings of SIGGRAPH 94, Computer Graphics Proceedings, Annual Conference Series, Andrew Glassner editor

(11). Cremer, James; Kearney, J.; Willemsen, P. (1996) “A Directable Vehicle Behavior Model for Virtual Driving Environments”, Proceedings of 1996 Conference on AI, Simulation, and Planning in High Autonomy Systems, La Jolla, CA.

(12).Pottinger, Dave (1999) “Coordinated Unit Movement” and “Implementing Coordinated Movement”, Game Developer Magazine, January, 1999.

(13). http://sourceforge.net/projects/opensteer/files/

(14). http://www.horde3d.org/

(15). http://www.collada.org

(16). http://sketchup.google.com/intl/it/product/gsup.html

(17). http://usa.autodesk.com/3ds-max/

(18). http://www.collada.org/mediawiki/index.php/ColladaMax

(19). Rita Beltrani, Sviluppo di un sistema per la simulazione grafica di folle in movimento, 26 gennaio 2010