

ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

SEDE DI CESENA

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI

CORSO DI LAUREA IN SCIENZE E TECNOLOGIE INFORMATICHE

**PROGETTAZIONE, MODELLAZIONE 3D E
SVILUPPO DI UN VIDEOGIOCO
MULTIPIATTAFORMA**

Relazione finale in Metodi Numerici per la Grafica

Relatore

Prof.ssa Damiana Lazzaro

Presentata da

Lorenzo De Donato

Anno Accademico 2011 – 2012

II Sessione

Indice

Introduzione	1
Capitolo 1: Progettazione	5
1.1 Descrizione e Scopo	5
1.2 Storyboard del gioco	6
1.3 Ambiente di visualizzazione	9
1.4 Funzionamento	10
1.5 Interazioni permesse all'utente.....	10
1.6 Difficoltà	10
1.7 Livelli / Scene.....	11
1.8 Punteggio e Tempo	11
1.9 Piattaforme e Game Engine	11
Capitolo 2: Unity.....	13
2.1 Introduzione al Game Engine.....	13
2.1.1 Storia	13
2.1.2 Cos'è Unity.....	13
2.1.3 Che tipo di piattaforma è possibile scegliere?.....	14
2.1.4 Che tipo di videogiochi è possibile creare?.....	14
2.2 Concetti base	14
2.2.1 Installazione.....	14
2.2.2 Requisiti di Sistema	15
2.2.3 Interfaccia.....	16

2.2.4 Scripting	22
2.3 Il motore di gioco	25
2.3.1 Rendering	25
2.3.2 Shader	26
2.3.3 Prestazioni	28
2.3.4 Illuminazione	30
2.3.5 Realtime Shadows.....	33
2.3.6 Screen Space Ambient Occlusion (SSAO)	34
2.3.7 Sun Shafts & Lens Effects.....	34
2.3.8 Lightmapping.....	35
2.3.9 Fisica	37
Capitolo 3: PlayMaker.....	39
3.1 Installazione	39
3.2 Concetti base sulla State Machine	40
3.3 Eventi	42
3.3.1 Introduzione.....	42
3.3.2 Eventi di sistema	43
3.3.3 Eventi Utente.....	44
3.4 Azioni.....	45
3.4.1 Introduzione alle Azioni.....	45
3.4.2 Ordine di esecuzione.....	46
3.4.3 Ciclo di vita di una Azione.....	46
3.5 Concetti base sull'Editing	48
3.5.1 Avviare Playmaker.....	48
3.5.2 Aggiungere una FSM ad un Game Object	49

3.5.3	Selezionare una FSM	49
3.5.4	Selezionare gli Stati	50
3.5.5	Impostare lo Stato di Start.....	51
3.5.6	Aggiungere Transizioni tra Stati.....	51
3.5.7	Aggiungere Azioni ad uno Stato.....	51
3.5.8	Variabili di State Machine.....	52
Capitolo 4: Modellazione		53
4.1	3D Studio Max	53
4.1.1	Introduzione.....	53
4.1.2	Interfaccia: Viste e Strumenti Principali	54
4.1.3	Le Primitive 3D Standard.....	57
4.1.4	Trasformazioni Base: Traslazione, Rotazione e Scalatura.....	58
4.1.5	Griglie e Unità di Misura	60
4.1.6	Pivot Point	62
4.1.7	Modificatori.....	63
4.2	Creazione di un modello	65
4.2.1	Introduzione.....	65
4.2.2	Modellazione del Ponte.....	66
4.2.3	Modellazione di un Animale	77
4.2.4	Modellazione, Pelle, Ossatura ed Animazione del Personaggio	86
4.3	Esportazione / Importazione del modello:	95
4.4	UV Mapping e Texturing.....	99
4.4.1	UV Mapping.....	99
4.4.2	Texturing	100
4.5	Render to Texture e LightMapping	104

Capitolo 5: Sviluppo.....	111
5.1 State Machine	111
5.1.1 Personaggio	111
5.1.2 Collider Manager	114
5.1.3 GUI Manager	115
5.1.4 Moduli Manager	116
5.1.5 Nemici Manager	117
5.1.6 Moduli Wrapper	118
5.1.7 Checkpoint.....	120
5.1.8 Bonus.....	121
5.1.9 Nemici – Edifici.....	121
5.2 Script e Algoritmi	124
5.2.1 Interazione col Personaggio	124
5.2.2 Randomizzazione degli Ostacoli	126
5.2.3 Start	129
5.2.4 Count Down.....	131
5.2.5 Pausa	134
5.2.6 Vite.....	140
5.2.7 Pausa Vite.....	143
5.2.8 Difficoltà	146
5.2.9 Punteggio.....	148
Capitolo 6: Il gioco in esecuzione.....	149
6.1 Configurazione	149
6.2 Menu di Partenza.....	150
6.3 Modalità di gioco a “Tempo”	151

6.3 Modalità di gioco a “Vite”	154
6.5 Pausa	155
Bibliografia	157
Ringraziamenti	159

Introduzione

Questa tesi nasce dall'esigenza di esplorare tutte le tematiche e le problematiche inerenti alla creazione di un videogioco, con lo scopo di realizzarne uno a partire dalla progettazione, modellazione e sviluppo su multiplatforma.

La scelta è stata dettata dalla passione personale nell'ambito della creazione di software multimediali, interattivi e per la grafica a computer, dalla personale esperienza in ambito lavorativo e dal desiderio di creare un progetto che affrontasse le fasi più importanti relative all'argomento.

L'avventura si svolge nello scenario urbano, composto da tre livelli principali: la campagna, la periferia ed il centro città. Ogni livello è caratterizzato da personaggi, oggetti ed edifici dalle forme bizzarre. Ogni oggetto ha un suo tema grafico e caratteristiche proprie, che spesso influiscono sulle dinamiche di gioco. Esistono due modalità di gioco: a "tempo" e a "vite". Nella prima l'obiettivo principale è quella di completare i livelli entro un limite di tempo prefissato. Nella seconda modalità l'obiettivo è quello di collezionare più punti possibile senza perdere le vite messe a disposizione.

Il motore di gioco utilizzato è Unity. La scelta è stata dettata dal fatto che Unity è molto diffuso nell'ambito dello sviluppo di piccoli/medi videogiochi e dal fatto che può pubblicare ed esportare su quasi qualsiasi piattaforma. Inoltre integra al suo interno un motore fisico, un motore audio e un *terrain editor* per costruire i terreni. Per queste ragioni è orientato alla progettazione di videogiochi.

Per implementare una parte dello sviluppo è stato usato il plug-in PlayMaker, un editor visuale orientato alla creazione di macchine a stati finiti. È stato scelto per la semplicità d'uso e per la facilità col quale è stato possibile programmare molti dei comportamenti usati nel gioco.

Il software di modellazione scelto è 3D Studio Max, un potente programma che permette la realizzazione di immagini, animazioni ed effetti speciali in computer grafica. È uno strumento che si rivela molto utile in vari campi (dall'architettura alla realizzazioni di effetti speciali od anche alla moda) soprattutto grazie

all'innumerabile numero di funzioni che ingloba (ben più di 3000) ed alla grande quantità di plug-in esterni che permettono di realizzare diverse tipologie di progetti in maniera sempre più veloce ed accurata.

Il videogioco si inserisce nel genere *platform game* (videogioco a piattaforme) ed è stato realizzato appositamente per piattaforme Windows e Mac. Si tratta di un videogioco tridimensionale. La meccanica di gioco implica l'attraversamento di livelli costituiti da piattaforme. Un *platform game* viene strutturato in modo tale che il personaggio si possa muovere dalla sinistra alla destra dello schermo con visuale in terza persona. L'eroe può saltare, evitare nemici e collezionare oggetti.

Ad oggi l'industria dei videogiochi di questo tipo è sempre più orientata ad un target d'età statisticamente collocato tra i 24 e i 44 anni. Infatti si pensa all'uso di un *platform game* in un contesto che ne preveda l'utilizzo, da parte dell'utente, durante piccoli ritagli di tempo quotidiani, con lo scopo di divertirsi ed anche rilassarsi. Ciò è dovuto anche dall'effetto trainante delle mini console portatili e dei dispositivi mobili come cellulari e tablet. Inoltre vale la pena sottolineare due aspetti non sempre debitamente analizzati e approfonditi: un primo relativo alla, per così dire, involontarietà educativa di molti giochi creati con l'unico scopo di divertire; un secondo relativo alle straordinarie valenze formative insite nella progettazione e creazione di videogiochi.

Lo sviluppo di videogiochi è un procedimento che implica un insieme di conoscenze non indifferente: si sta producendo un contenuto multimediale, ponendosi così allo stesso livello di "sviluppatori" di libri, di film, eccetera. In base al numero di persone che formano il team si avrà, o meno, la possibilità che la tutta conoscenza necessaria possa essere distribuita fra i componenti, anche se ciò non sempre risulta sufficiente. Andando ad analizzare quello che realmente serve per costruire un videogioco bisogna premettere un fattore: la tipologia che si intende sviluppare è una notevole discriminante al carico di lavoro che si dovrà sostenere.

Lo studio degli strumenti necessari alla progettazione è stato di fondamentale importanza per affrontare il lavoro di tesi. Uno degli obiettivi principali è stato quello di approfondire le conoscenze necessarie inerenti ai software usati. La

creazione di un videogioco infatti può coinvolgere diverse tipologie di programmi: fotoritocco, modellazione tridimensionale, *Game Engine* ed ambienti di programmazione.

Gli argomenti trattati includono la progettazione (*Game Design*), la modellazione ed, infine, lo sviluppo finale. Nella progettazione avviene l'ideazione delle regole e dei contenuti di un gioco e si tratta, ovviamente, della parte più creativa durante la quale si può dare sfogo alla propria fantasia. Molte volte è proprio questa fase che determina il successo o il fallimento del prodotto. In seguito si prosegue con la programmazione e le modellazioni grafiche per tradurre il progetto in una realtà concreta.

Nel primo capitolo vengono elencati gli step di progettazione necessari all'ideazione. Tra le fasi più importanti di questo processo è giusto citare quella dello *Storyboard*, durante la quale vengono disegnate a mano le scene dell'ambiente di visualizzazione selezionato e vengono decise le regole fondamentali per i comportamenti che si dovranno implementare nello sviluppo. Non di minore importanza risulta la scelta delle piattaforme di destinazione e del *Game Engine* (motore di gioco) come ambiente di sviluppo.

Nel secondo capitolo viene analizzato Unity, il motore di gioco utilizzato per lo sviluppo, l'importazione dei modelli e la creazione delle scene. Dopo un breve accenno alla storia, ai concetti base per il suo uso ed all'interfaccia è stato analizzato il sistema di scripting interno utilizzato per programmare, il funzionamento e le caratteristiche principali messe a disposizione dal prodotto.

Nel terzo capitolo è stato presentato PlayMaker, l'editor visuale di *State Machine* utilizzato per sviluppare alcuni comportamenti del videogioco. PlayMaker è una *Runtime Library* per Unity3D capace di rendere facile ed intuitivo l'utilizzo delle macchine a stati finiti utilizzate nella programmazione.

Nel quarto capitolo viene preso in esame il software scelto per la modellazione degli oggetti e delle scene: 3D Studio Max. Vengono presentati i concetti base per l'utilizzo e l'interfaccia grafica del programma. Vengono analizzate, inoltre, le tecniche di modellazione più importanti per la creazione degli oggetti.

Nel quinto capitolo sono state trattate le tematiche di sviluppo e programmazione del progetto. Vengono prese in esame le *state machine* usate e tutti i principali script che implementano i comportamenti del personaggio, degli oggetti e dell'interfaccia grafica.

Nell'ultimo capitolo vengono presentati alcuni *screenshot* del videogioco in esecuzione per mostrare una rappresentazione finale dell'intero lavoro, in particolare: l'interfaccia di configurazione iniziale, il menu di partenza, la modalità di gioco a "tempo", la modalità di gioco a "vite" e l'interfaccia del gioco in pausa.

Capitolo 1

Progettazione

In questo capitolo vengono elencate le fasi di progettazione necessarie per poter sviluppare un videogioco. Fra le fasi più importanti di questo processo è giusto citare quella dello *storyboard* durante la quale vengono disegnate a mano le scene dell'ambiente di visualizzazione scelto e vengono decise le regole fondamentali per i comportamenti che si dovranno implementare nello sviluppo. Non meno importante è la scelta delle piattaforme di destinazione e del *game engine* (motore di gioco) scelto come ambiente di sviluppo i quali, a seconda della scelta fatta, potrebbero influenzare il flusso di lavoro previsto.

1.1 Descrizione e Scopo

“Move It” è il nome del videogioco ovvero in italiano “Muovilo”! La scelta del nome non è a caso, infatti il videogioco è pensato per essere un gioco di abilità il più immediato possibile e per dare la possibilità all'utente di utilizzarlo senza il bisogno della minima istruzione. Vi sono due modalità di gioco: a “tempo” e a “vite”.

Nella modalità di gioco a “Tempo” si deve muovere il personaggio principale all'interno della scena prevista schivando gli ostacoli presenti per riuscire a completare il livello attuale.

É prevista anche una modalità a “Vite” in cui il player deve schivare gli ostacoli, pena la perdita di una vita.

Gli scenari creati e modellati sono: la campagna, la periferia ed il centro città.

Lo scopo del gioco è di avanzare evitando gli ostacoli nel livello scelto per poter arrivare all'obiettivo preposto. L'obiettivo finale nel caso della modalità di gioco a tempo è quello di finire il livello in un arco temporale massimo oppure, nel caso

della modalità a vite, è quello di ottenere il punteggio più alto possibile schivando gli ostacoli sul percorso.

1.2 Storyboard del gioco

Lo storyboard di un videogioco è un elemento indispensabile per progettare e visualizzare globalmente le fasi di gioco. Le azioni dei personaggi, i background nei quali essi si muovono e gli oggetti con cui interagiscono vengono illustrati prima di cominciare con la realizzazione allo scopo di focalizzare meglio i dettagli e avere un quadro generale dei momenti salienti. La realizzazione di uno storyboard richiede conoscenze di disegno e animazione nonché di un ottimo intuito utile a selezionare le scene più efficaci per dare l'idea di come si deve sviluppare il progetto.

Nel videogioco che è stato creato lo storyboard è stato utile per avere un'idea più precisa dello stile di modellazione del personaggio, delle ambientazioni e degli oggetti presenti e per capire come creare le azioni e le interazioni possibili.

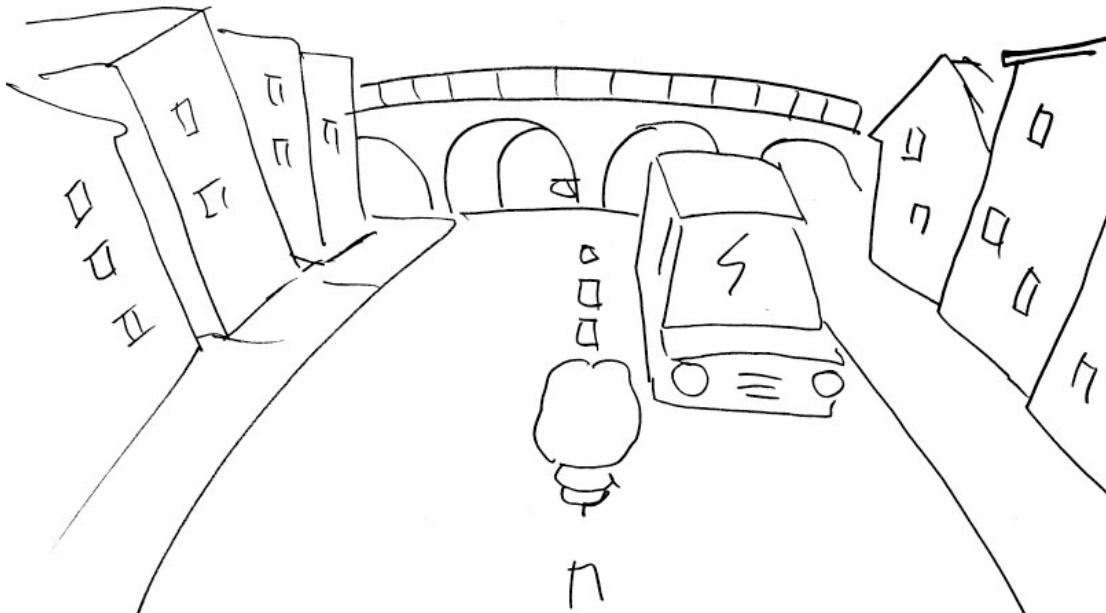


Fig. 1.1 Disegno della scena di gioco in periferia

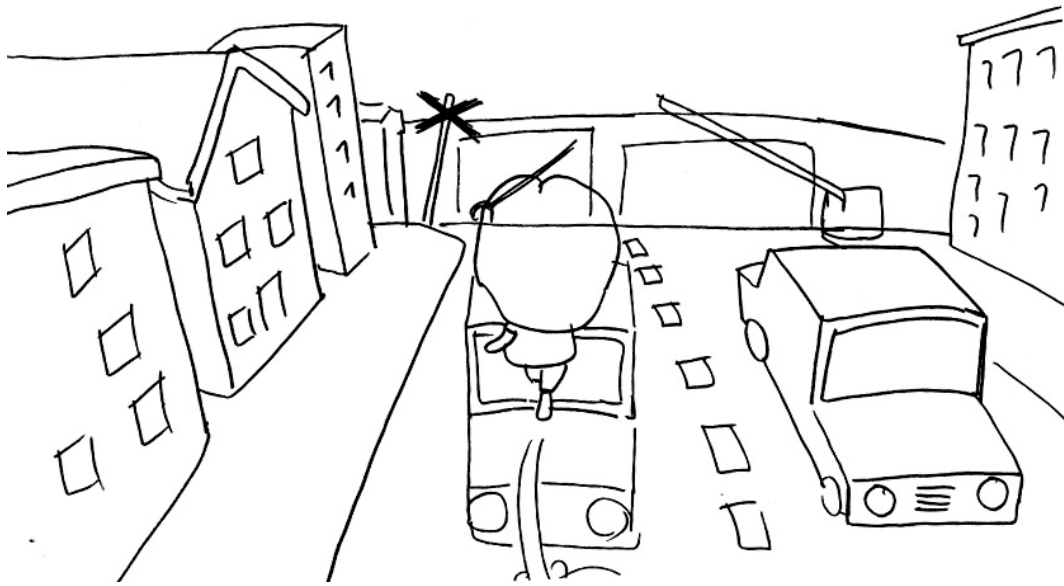


Fig. 1.2 Disegno del salto

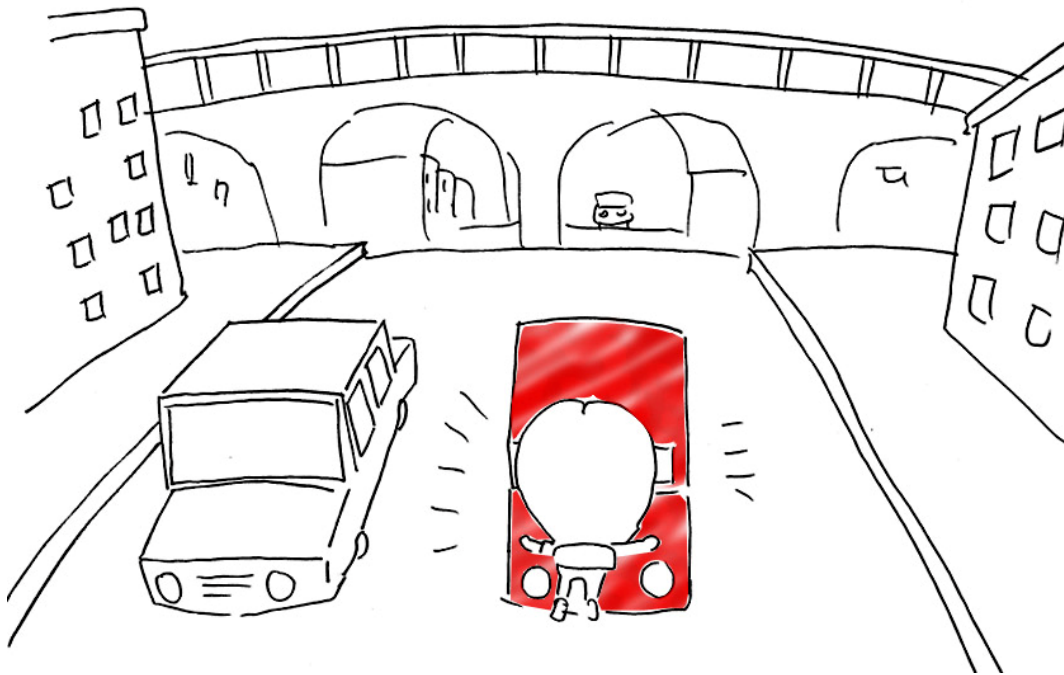


Fig. 1.3 Disegno della collisione con un ostacolo

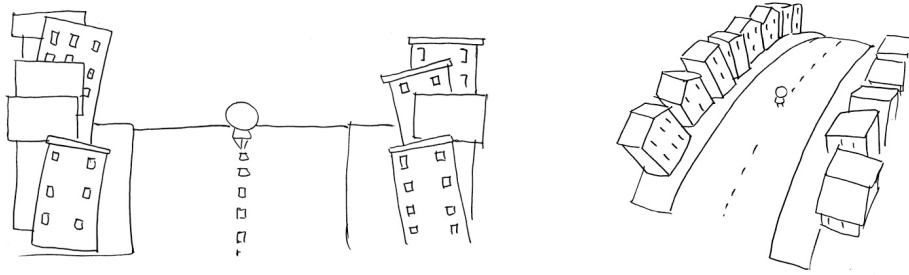


Fig. 1.4 Disegno del modulo principale: vista frontale e prospettiva

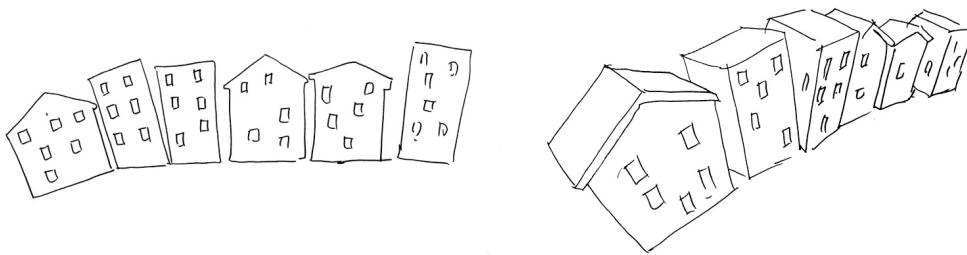


Fig. 1.5 Disegno delle case: vista frontale e prospettiva

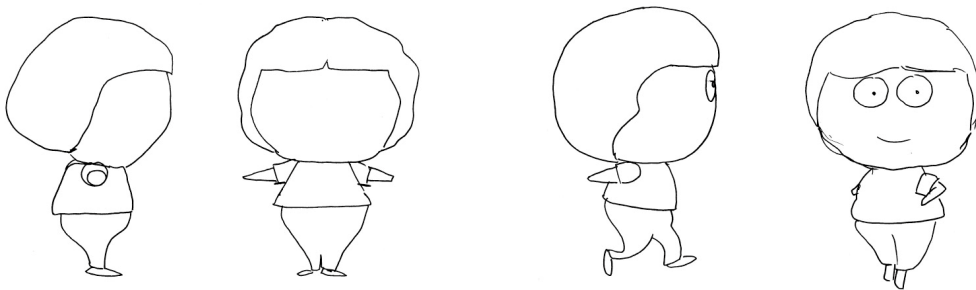


Fig. 1.6 Disegno del personaggio fermo e in movimento

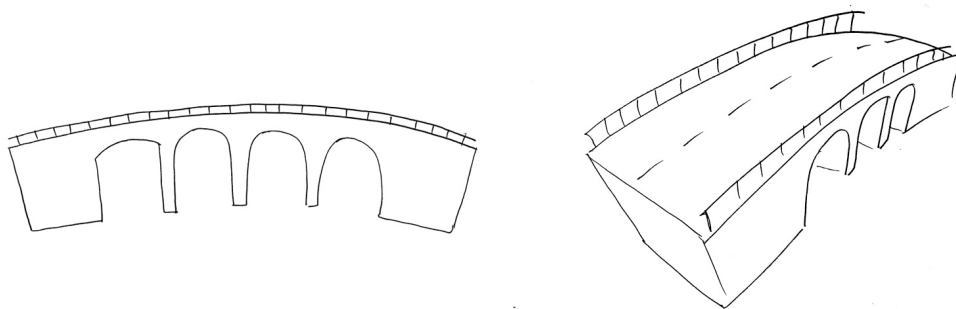


Fig. 1.7 Disegno del ponte: vista frontale e prospettiva

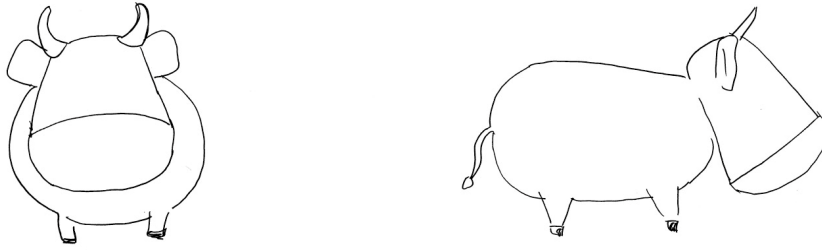


Fig. 1.8 Disegno dell'animale

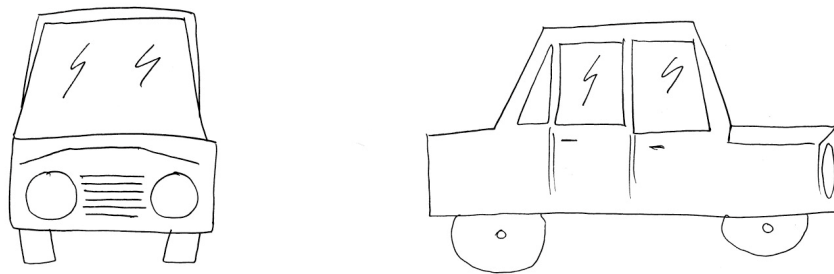


Fig. 1.9 Disegno della macchina

1.3 Ambiente di visualizzazione

Il gioco è stato pensato per essere visualizzato in un ambiente 3D. Lo scenario, rappresentato da case, edifici, alberi, oggetti e ostacoli vari presenti sulla scena, è stato creato attraverso il software di modellazione 3D Studio Max, mentre per quanto riguarda la creazione delle texture è stato usato il software di fotoritocco Photoshop.

Lo stile e il dettaglio di modellazione sono relativi alla grafica scelta per il videogioco, quindi gli oggetti sono stati modellati in stile “cartoon” senza comunque tralasciare una buona resa dell’illuminazione e degli effetti. Il processo di modellazione parte dall’uso dei disegni precedentemente creati nello storyboard che servono da guida per la creazione di ogni singolo modello relativo alle varie scene. La creazione dell’ambiente del videogioco prevede tecniche di diverso tipo che sono state utilizzate adottando anche l’uso di plug-in esterni.

1.4 Funzionamento

La camera collegata al personaggio è in terza persona, e l'effetto creato è quello di un avanzamento con una velocità costante. Lungo il percorso sono disposti alcuni ostacoli, che possono essere di due tipi:

- Ostacoli nemici: (per esempio persone, animali, automobili, ecc.)
- Oggetti della scena: (per esempio edifici, case, alberi, ecc.)

Se il personaggio principale entra in collisione con uno degli oggetti della scena definiti come ostacolo, l'avanzamento rallenta e l'interazione col personaggio si blocca per qualche attimo, per poi ripartire lentamente fino alla ripresa totale dell'interazione. Per evitare le collisioni l'utente deve quindi interagire con gli ostacoli; per l'utente l'interazione non cambia, è l'applicazione a distinguere il tipo di ostacolo e reagire di conseguenza: se l'utente interagisce con un ostacolo, l'avanzamento nella scena si ferma; se l'utente interagisce con un oggetto di tipo bonus l'avanzamento non rallenta ma aumenta il punteggio.

1.5 Interazioni permesse all'utente

- Input da tastiera: spostamento del personaggio usando “freccia destra” o “freccia sinistra” e salto del personaggio attraverso la pressione del tasto “barra spaziatrice”.
- Mouse: utilizzo del mouse per poter navigare attraverso l'interfaccia grafica del gioco.

1.6 Difficoltà

Il livello di difficoltà del videogioco è relativo al numero di elementi sulla scena anteposti all'avanzamento del personaggio. In dettaglio più saranno i “nemici” e gli “oggetti” che ostacoleranno il percorso dell'utente nello scenario più il livello di difficoltà aumenterà. In questo modo l'avanzamento del percorso nel gioco sarà responsabile dell'aumento del numero di ostacoli da evitare nel minor tempo possibile che comporterà una difficoltà maggiore.

1.7 Livelli / Scene

Lo scenario che è stato scelto è quello della città. In questo caso avremo tre livelli: il primo è la campagna , il secondo è la periferia mentre il terzo è il centro città. Per sviluppi futuri sono stati pensati altri scenari con vari livelli di gioco. Ogni livello è suddiviso a sua volta in moduli che verranno caricati a random attraverso algoritmi implementati negli script. Ogni modulo ha caratteristiche differenti come per esempio edifici di tipo diverso per poter differenziare la scena nel suo avanzamento.

1.8 Punteggio e Tempo

Il sistema di punteggio funziona in questo modo: nel gioco a “tempo” più veloce sarà l’utente a percorrere l’intera scena ed a finire il livello, maggiore sarà il punteggio ottenuto; nel gioco a “vite” più lunga sarà la durata del tempo di gioco più alto sarà il punteggio. Ad ogni ostacolo col quale si collide sarà associata una perdita del tempo a disposizione.

Il calcolo del punteggio finale sarà associato al tempo trascorso per completare il livello unito a vari bonus (prima completo il livello più alto sarà il punteggio).

Lo scorrere del tempo massimo è assoluto e non dipende dalla velocità di avanzamento dell’utente nello scenario, ma il calcolo del tempo ottenuto per completare un livello sarà influenzato anche dalla bravura dell’utente nell’evitare gli “ostacoli” e gli “oggetti” sul percorso.

1.9 Piattaforme e Game Engine

Il gioco è stato realizzato per piattaforme Win/Mac, ma è stato pensato per essere utilizzato anche su dispositivi touch screen. Si pensa ad uno sviluppo futuro su piattaforme mobili come IOS e Android.

Il motore di gioco scelto è Unity. La scelta è stata dettata dal fatto che Unity è un *game engine* molto diffuso nell’ambito dello sviluppo di piccoli/medi videogiochi e dal fatto che può pubblicare ed esportare per quasi qualsiasi piattaforma come ad esempio: Win/Mac, iPhone, Android, Nintendo Wii, PS3, ed anche Xbox. Questo

motore ha molti vantaggi infatti può, ad esempio, importare un modello creato in 3D Studio Max senza alcun problema ed aggiornare il modello con le modifiche fatte senza dover reimportare ogni volta. Inoltre integra al suo interno un motore fisico, un motore audio e un *terrain editor* per costruire i terreni, quindi è orientato alla progettazione di videogiochi.

Per poter realizzare il gioco inoltre è stato fatto uso di PlayMaker, il quale integra in Unity un'editor visuale per creare macchine a stati finiti ed un insieme di librerie a runtime pronte per l'uso. Grazie a questo strumento il flusso di lavoro è più snello e scorrevole ed inoltre è possibile progettare e implementare le varie funzionalità previste dalla progettazione del gioco senza troppi problemi e senza conoscere alcun linguaggio di programmazione.

Capitolo 2



Unity

In questo capitolo si prende in esame Unity, il motore di gioco scelto per lo sviluppo, l'importazione dei modelli e la creazione delle scene. Dopo un breve accenno alla storia, ai concetti base per l'uso ed all'interfaccia di questo ambiente vengono analizzati il sistema di scripting interno per programmare, il funzionamento e le caratteristiche principali messe a disposizione.

2.1 Introduzione al Game Engine

2.1.1 Storia

Il motore Unity è sviluppato da Unity Technologies. La prima versione di Unity fu presentata all'Apple Worldwide Developers Conference nel 2005. A giugno 2012 Unity ha raggiunto una comunità di oltre 1,2 milioni di sviluppatori registrati includendo grandi compagnie di pubblicazione, piccoli studi, studenti e amatori. Oltre che per i videogiochi il motore di Unity viene usato per simulazioni, visualizzazioni, a scopo educativo e applicazioni interattive ed inoltre può pubblicare formati per 12 tipi di piattaforme includendo desktop, console e dispositivi mobili.

2.1.2 Cos'è Unity

Unity è un motore di sviluppo completamente integrato e ricco di funzionalità per la creazione di contenuti 3D interattivi. Esso fornisce completa funzionalità per assemblare contenuti di alta qualità, ad alte prestazioni e per la pubblicazioni su più piattaforme. Unity è pensato per aiutare i piccoli sviluppatori, grandi e piccoli studi, multinazionali, studenti ed appassionati per ridurre drasticamente il tempo,

gli sforzi ed i costi di creazione di videogiochi. È possibile utilizzare Unity e allo stesso tempo giocare e modificare il videogioco creato mentre è in esecuzione. Una volta finito di crearlo, è possibile pubblicare il prodotto.

2.1.3 Che tipo di piattaforma è possibile scegliere?

Con Unity si può pubblicare il gioco sulle seguenti piattaforme: Mac OSX App, Windows Executable, Web Browsers (usando lo Unity WebGL), iPhone, iPad, cellulari e tablets con sistema Android, Wii, PS3 e Xbox 360.

2.1.4 Che tipo di videogiochi è possibile creare?

Unity supporta la creazione di quasi tutti i tipi di giochi immaginabili tra i quali: *Browser-based MMOGs, First-person shooters, Racing games, Real-time strategy games, Third-person shooters, Roleplaying games, Side-scrollers.*

2.2 Concetti base

2.2.1 Installazione

L'ultima versione di Unity (Unity 3.5.6) è disponibile a questo indirizzo web: <http://unity3d.com/unity/download/>. Attraverso il browser recarsi alla pagina appena elencata e premere sul pulsante blu "Download Unity 3.5.6". In questo modo è possibile scaricare il file di setup con estensione ".exe" per sistemi Windows e con estensione ".dmg" per sistemi OS X.

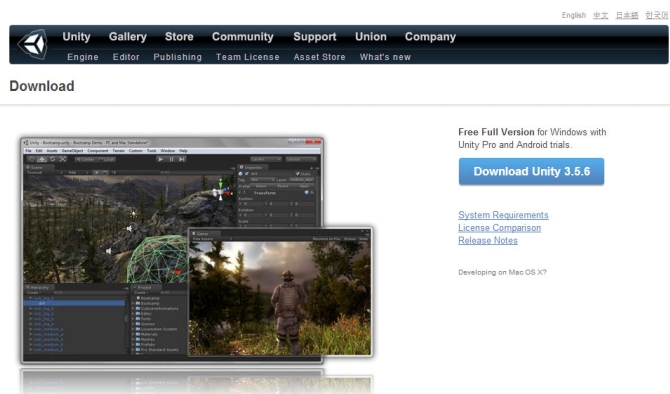


Fig. 2.1 Pagina di download di Unity

Una volta scaricato il programma localizzare la posizione del file di *setup* ed aprirlo. Il *setup* guiderà l'utente nel processo di installazione dando la possibilità di decidere in quale cartella installare Unity dopo aver accettato le condizioni di licenza che per Unity Pro prevedono un periodo di prova di trenta giorni. Finiti i trenta giorni di prova Unity passerà automaticamente alla versione base. Per leggere tutte le differenze tra la versione base e la versione Pro recarsi a questo indirizzo web: <http://unity3d.com/unity/licenses>.

2.2.2 Requisiti di Sistema

Sviluppo orientato a piattaforme Windows e OS X

- Windows: XP SP2 o versioni successive; Mac OS X: Intel CPU & "Leopard" 10.5 SP2 o versioni successive. Si noti che Unity non è stato testato su versioni server di Windows ed OS X.
- Schede grafiche con supporto alle DirectX 9 (shader model 2.0). Ogni scheda grafica prodotta dal 2004 in poi dovrebbe funzionare.
- L'uso dell'*Occlusion Culling* richiede un processore grafico (GPU) con supporto alle Query di Occlusione (alcune GPU Intel non supportano tale caratteristica).
- Il resto dipende dalla complessità effettiva del progetto realizzato.

Sviluppo orientato a piattaforme iOS

- Un computer Mac con processore Intel
- Mac OS X "Snow Leopard" 10.6 o versioni successive
- Il resto dipende dalla complessità effettiva del progetto realizzato.

Sviluppo orientato a piattaforma Android

- Windows XP SP2 o versioni successive; Mac OS 10.5.8 o versioni successive
- Android SDK e Java Development Kit (JDK)

- I contenuti per Android richiedono un device con le seguenti caratteristiche:
 - Android OS 2.0 o versioni successive
 - Dispositivo con processore ARMv7 (Cortex family)
 - Il supporto della GPU per le OpenGL ES 2.0 è raccomandato

Requisiti di sistema per contenuti sviluppati su Unity

- Windows XP o versioni successive; Mac OS X 10.5 o versioni successive
- Una scheda grafica abbastanza recente, dipende dalla complessità del gioco
- I giochi online vengono eseguiti su tutti i *browser* inclusi Internet Explorer, Firefox, Safari e Chrome oltre a tutti gli altri

2.2.3 Interfaccia

La finestra principale di Unity è composta da varie schede, chiamate Viste. Ogni vista ha il suo scopo specifico che sarà descritto nelle prossime sezioni [2].

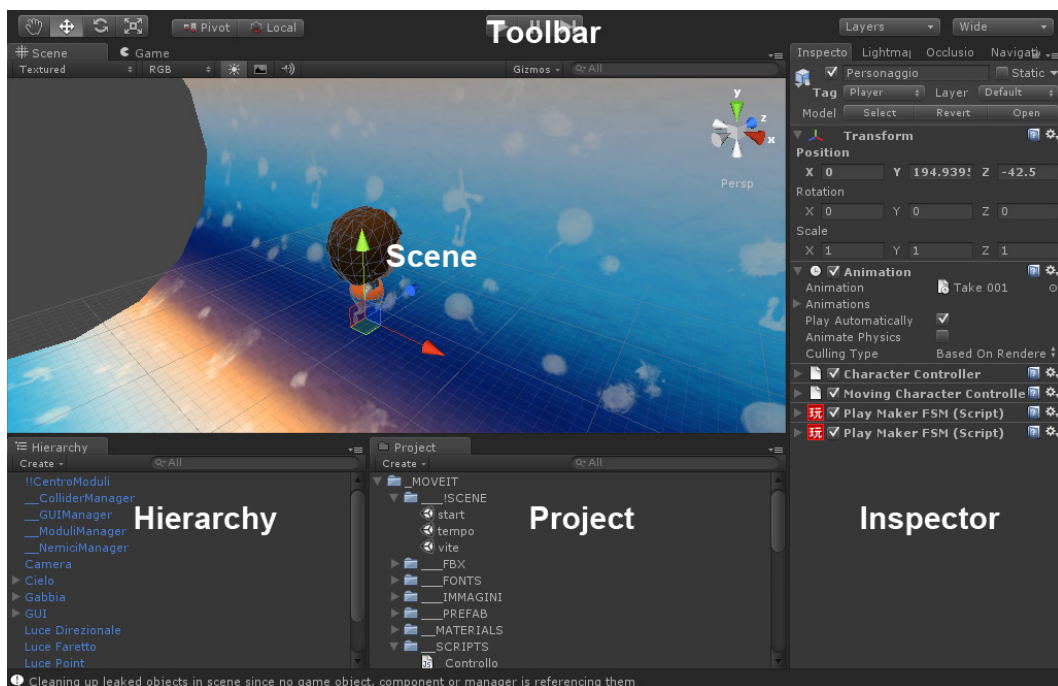


Fig. 2.2 Interfaccia Principale di Unity

Project View

Ogni progetto di Unity contiene una cartella *Assets* (attività). Il contenuto di questa cartella è rappresentato nella *Project View* (finestra di progetto). È qui che si memorizzano tutti gli *Assets* che compongono il gioco, come scene, script, modelli 3D, texture, file audio, e *Prefabs* (prefabbricati). Se si preme su uno qualsiasi degli *Assets* nella finestra di progetto, è possibile scegliere *Reveal in Explorer / Reveal in Finder on Mac* (Mostra in Esplora risorse / Mostra nel Finder su Mac) per vedere effettivamente dove è posizionato l'*Asset* nel file system.

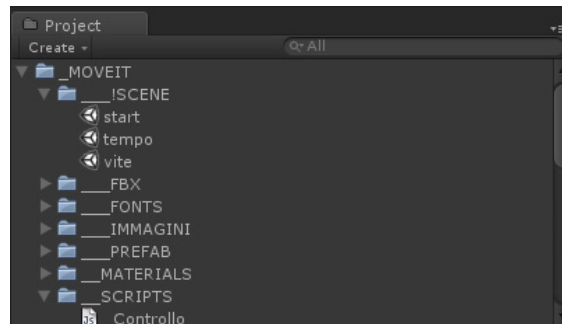


Fig. 2.3 Project View

Hierarchy View

La *Hierarchy View* (finestra gerarchia) contiene ogni *GameObject* presente nella scena. Alcuni di questi sono istanze di *Assets* come i modelli 3D, e altri sono esempi di prefabbricati, oggetti personalizzati che compongono gran parte del gioco. È possibile selezionare gli oggetti della gerarchia e trascinare un oggetto su un altro e fare uso di *Parenting* per inglobare un oggetto dentro l'altro. Quando gli oggetti sono aggiunti e rimossi nella scena, essi appaiono e scompaiono pure dalla Gerarchia.

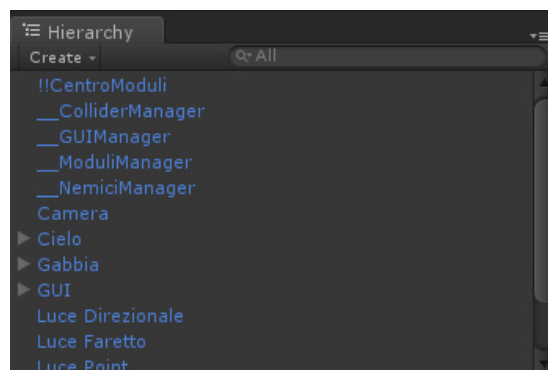


Fig. 2.4 Hierarchy View

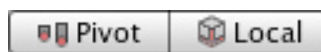
Toolbar

La barra degli strumenti è composta da cinque controlli di base, ognuno dei quali si riferisce a diverse parti dell'editor.

La *Transform Tools* viene usata nella finestra Scene per traslare, ruotare e scalare.



La *Transform Gizmo Toggles* serve per spostare il pivot dell'oggetto e per passare dalle coordinate locali a quelle di mondo. Viene sempre usata nella finestra Scene.



I pulsanti *Play/Pause/Step* vengono usati nella *Game View*.



Il *Layers Drop-down* controlla quali oggetti vengono visualizzati nella finestra Scene.



Il *Layout Drop-down* controlla la disposizione di tutte le *View*.



Scene View

La *Scene View* (finestra della scena) rappresenta l'accesso alla scena interattiva. Si utilizzerà la vista della scena per selezionare e posizionare gli ambienti, il giocatore, la camera, i nemici, e tutti gli altri *GameObjects*. Manovrare e manipolare gli oggetti all'interno della Scena sono alcune delle funzioni più importanti in Unity e a tal fine Unity fornisce una sequenza di tasti per le operazioni più comuni, ad esempio cliccando un *GameObject* e premendo il tasto F da tastiera la Scena e il Pivot verranno centrati sull'oggetto.

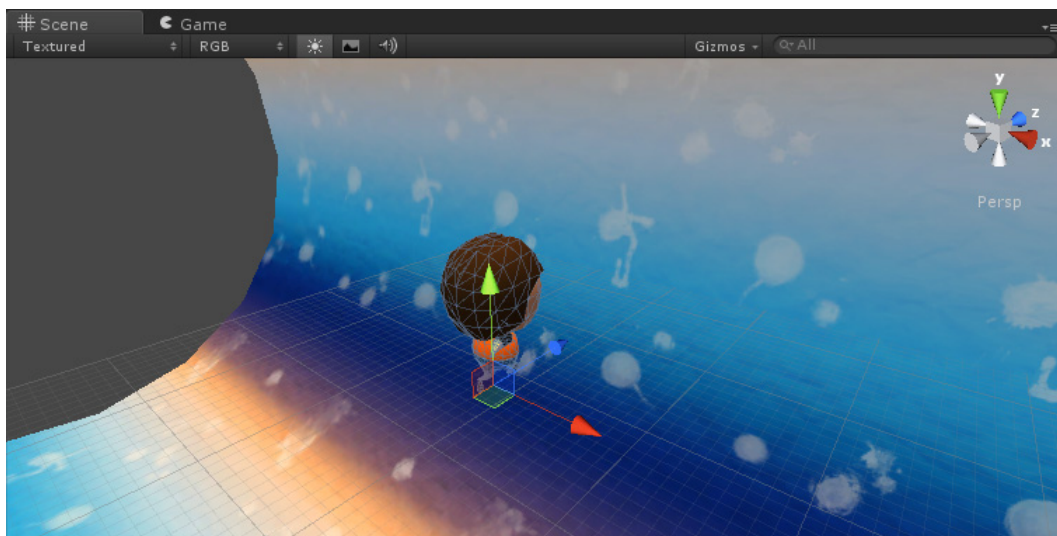


Fig. 2.5 Scene View

Nell'angolo in alto a destra della vista scena si trova il *Gizmo* della scena. Questo mostra l'orientamento attuale della Camera nella scena, e consente di modificare rapidamente l'angolo di visualizzazione. Ciascuno dei colori del gizmo rappresenta un asse geometrico. È possibile fare clic su uno degli assi per impostare la Camera ad una vista ortogonale (cioè, prospettiva-free) allineata lungo l'asse corrispondente. È possibile fare clic sul testo sotto il gizmo per passare dalla vista in prospettiva normale ad una vista isometrica.

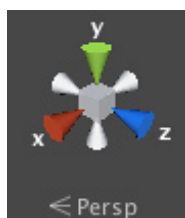


Fig. 2.6 Gizmo

Per traslare, ruotare e ridimensionare i singoli *GameObject* della scena utilizzare gli strumenti di trasformazione nella barra degli strumenti. Ciascuno ha un *Gizmo* corrispondente che appare attorno al *GameObject* selezionato. È possibile utilizzare il mouse e manipolare ogni asse del *Gizmo* per modificare la Trasformazione del *GameObject*, oppure è possibile digitare i valori direttamente nei campi numero del componente di trasformazione nella vista *Inspector*.

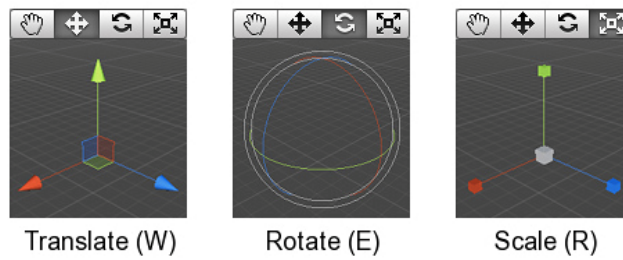


Fig. 2.7 Trasformazioni del GameObject

La *Scene View control bar* consente di vedere la scena in varie modalità di visualizzazione: *Textured*, *Wireframe*, *RGB*, *Overdraw* e molti altri. Essa consentirà inoltre di vedere (e udire) l'illuminazione, gli elementi di gioco, e i suoni nella scena.



Fig. 2.8 Scene View control bar

Inspector

I giochi in Unity sono costituiti da diversi tipi di *GameObject* che contengono mesh, script, suoni o altri elementi grafici come le luci. La vista *Inspector* visualizza informazioni dettagliate sul *GameObject* attualmente selezionato, inclusi tutti i componenti collegati e le loro proprietà. In questa vista è possibile modificare la funzionalità del *GameObject* nella scena.

Qualsiasi proprietà che viene visualizzata nella finestra di ispezione può essere modificata direttamente. Anche le variabili degli script possono essere modificate senza modificare lo script stesso. È possibile utilizzare l'*Inspector* per cambiare le variabili in fase di esecuzione e per sperimentare diversi comportamenti. In uno script, se si definisce una variabile pubblica di un tipo di oggetto (come ad *GameObject* o *Transform*), è possibile poi trascinare e rilasciare un *GameObject* o un *Prefab* nella finestra per assegnare l'oggetto a quella variabile. Infine è possibile utilizzare il menù a discesa *Layer* per assegnare un livello di rendering diverso ad un *GameObject*. Usare il menu a discesa *Tag* per assegnare un *tag* diverso ad un *GameObject*.

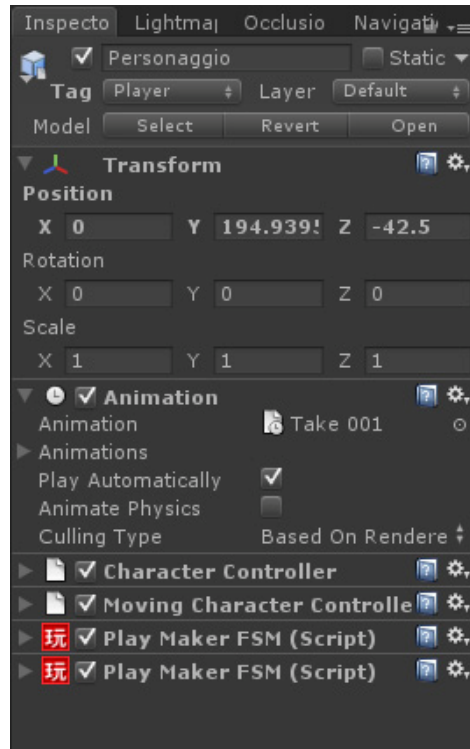


Fig. 2.9 Inspector

Game View

La *Game View* (visuale di gioco) è mostrata dalla Camera presente nel gioco ed è fondamentale per produrre la sua rappresentazione finale e pubblicarlo. Sarà necessario quindi utilizzare una o più camere per controllare ciò che il giocatore vede effettivamente quando sta giocando .



Fig. 2.10 Game View

Utilizzare i pulsanti nella barra degli strumenti per il controllo della modalità *Play* e per vedere come funzionerà il gioco una volta pubblicato. In modalità *Play*, tutte le modifiche apportate sono temporanee, e saranno resettate una volta rimesso in stop.



Fig. 2.11 Pulsanti Play Mode

Il primo elenco a discesa sulla barra di controllo del gioco è *Aspect*. Qui, è possibile forzare le proporzioni della finestra di visualizzazione del gioco su valori diversi. Può essere usato per testare come sarà su monitor con proporzioni diverse. Più a destra vi è il pulsante *Maximize on Play toggle*. Una volta attivato questo strumento la visuale di gioco sarà impostata al 100% rispetto alla finestra dell'editor per un'anteprima a schermo intero quando si è nella modalità di riproduzione. Proseguendo a destra vi è il *Gizmos toggle* (selettore del Gizmo). Con lo strumento attivato, tutti i *Gizmo* che appaiono nella vista scena saranno visualizzati nella scena del gioco. Infine abbiamo il pulsante *Stats* (statistiche). Questo pulsante, se attivato, mostra le statistiche del Rendering utili per l'ottimizzazione delle prestazioni grafiche.



Fig. 2.12 Game View Control Bar

2.2.4 Scripting

Linguaggi

Il sistema di scripting è basato su *Mono*, una versione *open-source* di *.NET*. Come *.NET*, *Mono* supporta molti linguaggi di programmazione, ma Unity supporta solo *C#*, *Boo*, e *Javascript*. Ogni linguaggio ha i suoi vantaggi e svantaggi. *C#* ha il vantaggio di avere una definizione ufficiale e molti libri didattici e *tutorial on-line*, oltre che ad essere simile ad un linguaggio principale come *.NET* e *Mono*, e quindi viene utilizzato in un gran numero di software commerciali ed *open-source*. Di contro *C#* è la scelta più ridondante. *Boo* sembra essere ben voluto dai programmatori *Python*, ma è probabilmente utilizzato dal minor numero di utenti Unity e quindi è il linguaggio con il minimo supporto. La versione di *Javascript*

in Unity è più ristretta ma a quanto pare deriva dal *Boo* e non è proprio come il *Javascript* standard (per cui è stato suggerito che possa essere chiamato *UnityScript*). La maggior parte degli script forniti nella documentazione sono in *Javascript*, anche se alcuni sono in *C #* [3].

Gli script sono Assets

Gli script sono *Assets*, proprio come texture e modelli e tanti altri tipi di oggetti. Diversi script sono forniti di base nella cartella *Standard Assets*. Gli script possono essere importati come tutti gli altri *asset* dal menu *Asset -> Import*.

Gli script sono Component

Unity è programmato tramite script - non vi è alcuna programmazione C++ in questione (a meno che non si scrivano plug-in o si compri una licenza dei file sorgente). Lo *Scripting* in Unity è un pò diverso dalla programmazione tradizionale, ma simile all'utilizzo del linguaggio di scripting *Linden* in *Second Life* e *Lua* in *CryEngine* ove singoli script sono attaccati agli oggetti nel gioco. Ciò si presta ad una progettazione orientata agli oggetti in cui la scrittura degli script serve per controllare particolari entità. In Unity, gli script sono componenti e fanno parte degli oggetti del gioco, proprio come gli altri componenti. In particolare, gli script fanno parte della classe *MonoBehaviour*, una sottoclasse diretta di *Behaviour*, il quale è un componente che può essere abilitato / disabilitato.

```
Object
  AnimationClip
  AssetBundle
  AudioClip
  Component
  Behaviour
    Animation
    AudioListener
    AudioSource
    Camera
    ConstantForce
    GUIText
    GUITexture
  GUILayer
  LensFlare
  Light
  MonoBehaviour
```

Fig. 2.13 La classe Object

Gli script sono classi

Ogni script definisce in realtà una nuova sottoclasse di *MonoBehaviour*. Con uno script *Javascript*, Unity considera questa definizione come implicita - la nuova classe è denominata in base al nome del file, e il contenuto dello script è trattato come se fosse all'interno di una definizione di classe.

Script Struttura

Un “pezzo” di codice associato a un oggetto non è molto interessante, se poi non viene eseguito. Gli script aggiungono comportamenti agli oggetti mediante l'attuazione di funzioni di *callback* che vengono richiamati dal motore Unity durante la vita di un gioco. Qui di seguito vengono elencate le più importanti.

- *Awake*: viene chiamata quando l'oggetto viene caricato.
- *Start*: viene chiamata subito dopo *Awake*, ma solo quando o se l'oggetto è attivo (e solo dopo la prima attivazione, non se lo stato dell'oggetto attivo viene ripetutamente acceso o spento).
- *Update*: viene chiamata fotogramma per fotogramma mentre l'oggetto è attivo, dopo lo *Start*.
- *FixedUpdate*: viene chiamata dopo un intervallo di aggiornamento che viene fissato a priori (lo stesso intervallo usato per gli aggiornamenti della fisica).

A volte si desidera eseguire azioni quando un oggetto di gioco è attivato o disattivato. Ad esempio, si potrebbe volere che la tabella dei punteggi di un gioco venga visualizzata ogni volta che debba essere attivata. Se si implementa quel codice nella funzione di *Start* sarebbe stato eseguito solo una volta al massimo. Proprio per questo è necessario usare la funzione *OnEnable* e se vi è necessita esiste anche la funzione inversa *OnDisable*.

Altre funzioni vengono richiamate solo per certi tipi di eventi attivati dalla fisica:

- *OnCollisionEnter*
- *OnCollisionStay*
- *OnCollisionExit*

2.3 Il motore di gioco

2.3.1 Rendering

Unity supporta diversi tipi di *Rendering*. La scelta di uno o dell'altro dipende dal tipo di videogioco che si sta creando e dal tipo di piattaforma e di hardware che si intende usare. Ogni *Rendering Path* ha differenti caratteristiche e prestazioni diverse che possono interagire con le luci e le ombre [6].

Il *Rendering Path* da usare dovrà essere selezionato in *Player Settings*. Se la scheda grafica non può supportare il *rendering path* selezionato Unity ne userà automaticamente uno con una qualità minore. In questo modo in una GPU che non può supportare il *Deferred Lighting* verrà usato il *Forward Rendering*. Se il *Forward Rendering* non è supportato verrà usato *Vertex Lit*.

Deferred Lighting

Il *Deferred Lighting* è il *rendering path* con la miglior qualità di illuminazione e di creazione delle ombre. È la scelta migliore se abbiamo un gran numero di luci in realtime, ma richiede un certo livello di hardware infatti non è supportato su dispositivi mobile. Per maggiori dettagli consultare la pagina: <http://docs.unity3d.com/Documentation/Components/RenderTech-DeferredLighting.html>.

Forward Rendering

Forward è un *rendering path* basato sugli *shader*. Supporta l'illuminazione pixel per pixel (incluso le *normal map*) e la creazione di ombre in *realtime* da una luce direzionale. Nelle impostazioni di default vengono calcolate un numero limitato di luci pixel per pixel, mentre il resto delle luci sono calcolate sui vertici dell'oggetto. Nel *Forward Rendering*, le luci più vicine all'oggetto sono completamente renderizzate attraverso la modalità di illuminazione per-pixel (tecnica in cui l'immagine della scena calcola l'illuminazione per ogni pixel che viene renderizzato [1]). Dal quarto punto di luce in poi vengono calcolate per-vertex (tecnica che calcola l'illuminazione di un modello 3D per ogni vertice e poi interpola i valori risultanti sopra le facce del modello per calcolare i valori finali

del colore per ogni pixel [1]). Le altre luci sono calcolate attraverso la tecnica *Spherical Harmonics* che è più veloce ma è solo un'approssimazione. Per maggiori dettagli consultare la pagina:

<http://docs.unity3d.com/Documentation/Components/RenderTech-ForwardRendering.html>.

Vertex Lit

Vertex Lit è il *rendering path* con il livello di qualità di illuminazione più basso e non vi è supporto per le ombre in *realtime*. È la miglior scelta su hardware limitato o su piattaforme mobili che non supportano gli altri tipi di rendering. Per maggiori dettagli consultare la pagina:

<http://docs.unity3d.com/Documentation/Components/RenderTech-VertexLit.html>.

2.3.2 Shader

Unity viene fornito con più di 100 *shader*¹ che vanno dai più semplici (*Diffuse*, *Glossy*, ecc) a quelli molto più avanzati (*Self-Illuminated*, *Bumped specular*, ecc.). Unity include più di 40 tipi *shader* ed è possibile aggiungerne degli altri. Gli *shader* in Unity sono creati attraverso il pannello *Materials*, che essenzialmente combina il codice degli *shader* con altri parametri come texture. Le proprietà del Materiale appariranno nel pannello *Inspector* quando un *Material* o quando un *GameObject* che contiene un materiale viene selezionato.

¹ La parola inglese *shader* indica uno strumento della computer grafica 3D che generalmente è utilizzato per determinare l'aspetto finale della superficie di un oggetto. Consiste essenzialmente in un insieme di istruzioni. Gli *shader* devono riprodurre il comportamento fisico del materiale che compone l'oggetto cui sono applicati. Si può quindi creare uno *shader* per i metalli, uno per la plastica, uno per il vetro e così via, e riutilizzarli più volte all'interno di una scena. Una volta modellato un oggetto complesso, come può essere ad esempio una finestra, si associa al modello della cornice uno *shader* per il legno, uno per la maniglia, e uno per il vetro. La caratteristica riutilizzabilità di questo strumento è preziosa nel lavoro con la computer grafica 3D, sia in termini di tempo che di risultato finale. Per determinare l'apparenza della superficie, essi utilizzano tecniche già consolidate come l'applicazione di texture, gestione delle ombre. Gli *shader* possono anche essere usati per applicare effetti di postprocessing. Essendo programmi a tutti gli effetti, è possibile utilizzarli anche per la replicazione di eventi fisici molto complessi quali collisioni e simulazioni fluidodinamiche.

Il pannello *Material* appare così:

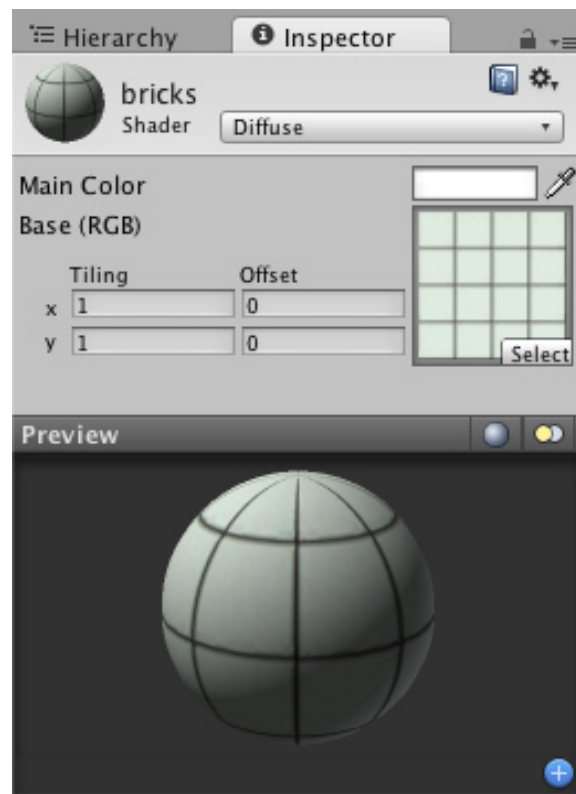


Fig. 2.14 Pannello Material

Ogni materiale apparirà in modo diverso nell'*Inspector* dipendentemente dallo specifico *shader* che si sta usando. Lo *shader* determina quale tipo di proprietà sono disponibili per le modifiche nel pannello *Inspector* per quel tipo di materiale. Bisogna ricordare che uno *shader* viene implementato attraverso l'uso di un materiale. Così mentre gli *shader* definiscono proprietà che vengono mostrate nell'*Inspector*, ogni materiale contiene i dati riguardanti slider, colori e texture. La cosa più importante da ricordare è che un singolo *shader* può essere utilizzato in più Material, ma un singolo Material non può utilizzare più di uno *shader*.

Scalabilità

Quando si utilizzano effetti di *shader* avanzati si vuole assicurare che il gioco funzioni bene su qualsiasi hardware di destinazione. Il sistema di *shader* di Unity fa ampio uso di un sistema di *fallback* in modo che ogni utente possa avere la migliore esperienza possibile e per questo supporta anche l'emulazione della scheda grafica per semplificare i test.

Post-processing

Unity ha un gran numero di effetti di post-elaborazione delle immagini a tutto schermo tra i quali troviamo: riflessi di luce attraverso gli alberi, profondità di campo ad alta qualità, effetti di lente, aberrazione cromatica, curve di correzione del colore e molto altro.

Surface Shader

Con Unity 3, sono stati introdotti i *Surface Shaders*, un nuovo metodo semplificato per costruire *shader* per più tipi di dispositivi e *rendering paths*. Abbiamo la possibilità di scrivere un semplice programma e Unity compilerà sia per *Forward & Deferred Rendering* rendendolo funzionante con le *lightmaps* e convertendolo automaticamente in GLSL per i dispositivi mobili.

Questo significa che è possibile concentrarsi su come ottenere il look giusto e fare in modo che possa adattarsi fino ad usare le *Spherical Harmonics* o funzioni fisse di illuminazione su hardware di bassa fascia. Se si desiderano modelli di illuminazione personalizzati basta applicare l'equazione di illuminazione all'interno di una funzione e Unity fa tutto il resto per qualsiasi metodo di rendering.

2.3.3 Prestazioni

Batching

Per ridurre al minimo le chiamate di rendering, Unity combinerà automaticamente la geometria delle scena in batches. Questo minimizzerà in modo significativo i sovraccarichi mantenendo comunque la massima flessibilità. Unity, inoltre, combina insieme oggetti statici in fase di compilazione per assicurare la massima capacità di elaborazione della geometria.

Occlusion Culling

Insieme a *Umbra Software* Unity ha sviluppato una tecnica completamente nuova per quanto riguarda soluzioni di visibilità precalcolate. L'*Occlusion Culling* funziona su cellulari, web e console



con sovraccarico minimo a livello di runtime, riducendo il numero di oggetti

renderizzati a quelli necessari. Viene usata per determinare quale superficie e quali parti delle superfici non dovrebbero essere visibili all'utente da un certo punto di vista e che non verranno renderizzate. Considerando i vari passi riguardanti la *rendering pipeline*, la proiezione, il piano di *clipping* e la rasterizzazione viene implementata attraverso l'uso dei seguenti algoritmi:

- *Z-buffering*
- *Scan Line Edge List*
- *Depth Sort* (Algoritmo del Pittore)
- *Binary space partitioning* (BSP)
- *Ray tracing*

Per maggiori informazioni sull'*Occlusion Culling* consultare la pagina web:

<http://docs.unity3d.com/Documentation/Manual/OcclusionCulling.html>.

GLSL Optimizer

Le *OpenGL ES* consentono di usare gli *shader* su dispositivi mobili. Purtroppo molti driver grafici lasciano ancora a desiderare, così è stato sviluppato un ottimizzatore per le *GLSL shader* con un miglioramento di 2-3 volte del fillrate² della scheda video.

LOD Support

Come le scene diventano più complesse le prestazioni sono da tenere sempre più in considerazione. Un modo per gestire questa situazione è quello di avere mesh con differenti livelli di dettaglio a seconda di quanto la camera è lontana dall'oggetto. Unity può gestire la situazione usando *LOD (level of detail) Groups* grazie al supporto interno sul livello di dettaglio da applicare all'oggetto.

² Il Fillrate è un parametro per valutare le prestazioni di una scheda video. Esistono due tipologie di fillrate che vengono prese in considerazione: pixel fillrate - indica la quantità di pixel che la GPU (Graphics Processing Unit) è in grado di scrivere nella memoria video in un secondo e viene misurato in Megapixel/s. Esso corrisponde al prodotto tra la frequenza di clock del processore grafico e il suo numero di pipeline; texel fillrate - indica la quantità di texel (elemento fondamentale di una texture) visualizzabili in un secondo e viene misurato in Megatextel/s.

Attraverso questo tipo di tecnica avremo la possibilità di usare due tipi di *mesh*: una ad alta risoluzione da utilizzare quando la camera è più vicina e l'altra a bassa risoluzione da utilizzare quando siamo lontani. In questo modo le prestazioni saranno sicuramente migliori.

2.3.4 Illuminazione

Il motore di rendering interno di Unity supporta il *Linear Space Lighting* (correzione di gamma) e il *rendering HDR*. La resa dell'illuminazione è ancora più veloce in Unity grazie al nuovo motore di *render multi-thread*.

Il Linear Lighting

Si riferisce al processo di illuminazione di una scena che abbia tutti gli ingressi linearizzati. Normalmente le textures sono create con una gamma pre-applicata e ciò significa che quando vengono utilizzate nei materiali non saranno lineari. Se queste texture verranno utilizzate nell'equazione di illuminazione globale porteranno a risultati non corretti del calcolo, poichè ci si aspetta di avere ingressi linearizzati prima dell'uso. Quindi grazie al processo di linearizzazione possiamo essere sicuri del fatto che sia gli ingressi che le uscite di uno shader siano nella corretta gamma di colori e questo è il risultato di un'illuminazione corretta che dovrebbe rispecchiare meglio la realtà.

Linear Intensity Response

Quando si sta usando il *Gamma Space Lighting* i colori e le texture interne ad uno shader hanno una correzione di gamma. Infatti i colori usati in uno shader con un'alta illuminazione sono più brillanti e luminosi di quello che dovrebbero essere nell'illuminazione lineare. Questo significa che più la luce cresce più la superficie diventerà brillante in maniera non lineare. In questo modo l'illuminazione potrebbe essere troppo forte in alcuni punti e potrebbe anche restituire modelli e scene non corrispondenti al reale. Quando si usa il *Linear Lighting* succede che la risposta della superficie rimarrà lineare anche se cresce l'intensità della luce. Questo tecnica porta ad avere superfici più realistiche e una miglior resa dei colori nella scena di gioco dando la possibilità di regolare l'illuminazione in maniera più realistica.

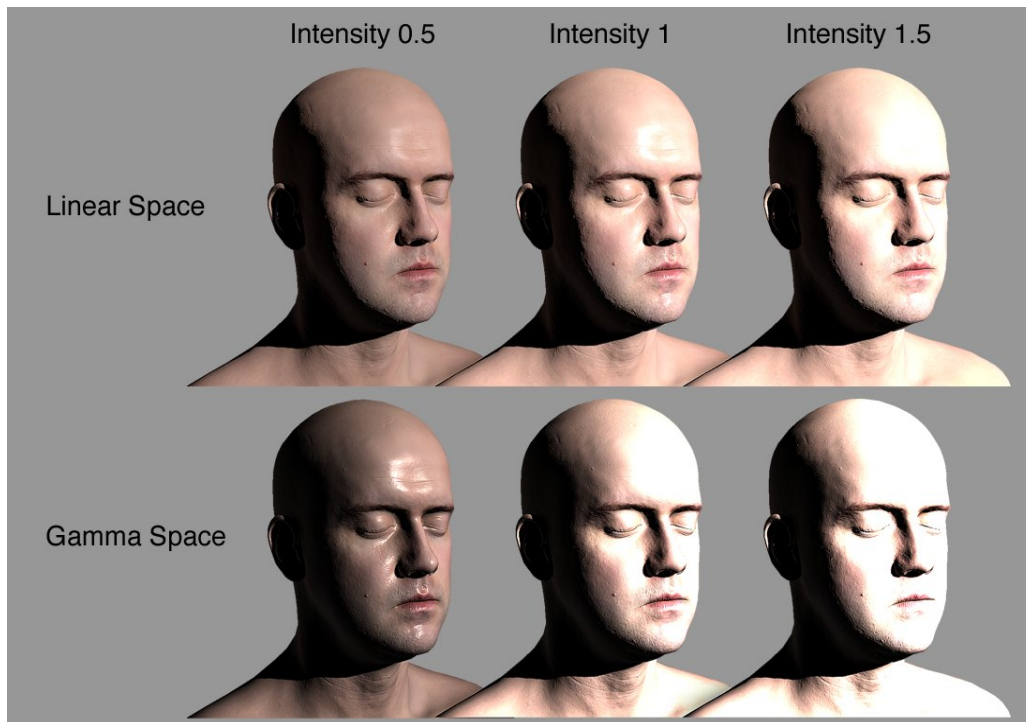


Fig. 2.15 Infinite, 3D Head Scan creato da Lee Perry-Smith.

Come usare il Linear Lighting

Attivare il linear lighting è molto semplice. Questa caratteristica può essere implementata in ogni progetto: *Edit > Project Settings > Player > Other Settings*.



Fig. 2.16 Other Settings

Lightmapping

Quando si utilizza il linear lighting tutte le luci e le texture vengono linearizzate, questo significa che i valori che verranno passati al lightmapper hanno bisogno di essere modificati. Cambiando il tipo di illuminazione è necessario rifare il *bake* delle *lightmap*. L'interfaccia di Unity mostra un avviso se le lightmap non si trovano nello spazio di colori corretto.

HDR (High Dynamic Range)

Nel rendering standard i valori del rosso, del verde e del blu per ogni pixel sono ognuno rappresentato in un range compreso tra 0 e 1 dove 0 rappresenta l'intensità minima, mentre 1 rappresenta l'intensità massima per lo schermo del dispositivo. Questo concetto è semplice e facile da applicare, ma non riflette con precisione il modo in cui funziona l'illuminazione in una scena di vita reale. L'occhio umano tende ad adattarsi alle condizioni di illuminazione locali, poichè un oggetto che appare bianco in ambiente poco illuminato può apparire meno luminoso di un oggetto che sembra grigio in pieno giorno. Inoltre, l'occhio è più sensibile alle differenze di luminosità nella parte bassa del range rispetto alla fascia alta.

Gli effetti visivi più convincenti possono essere ottenuti se la resa è atta a permettere gli intervalli di valori di pixel che riflettono più accuratamente i livelli di luce che sarebbero presenti in una scena reale. Permettere alla rappresentazione interna della grafica di utilizzare i valori al di fuori del range 0..1 è l'essenza del rendering in *High Dynamic Range* (HDR). HDR può essere attivato separatamente per ogni camera presente nella scena usando le impostazioni nel pannello del componente Camera:

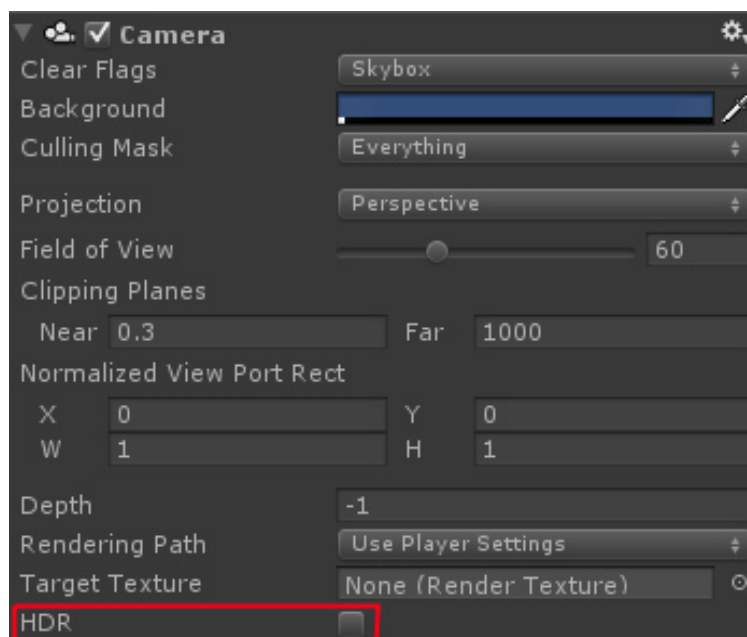


Fig. 2.17 Componente Camera

2.3.5 Realtime Shadows

Le ombre in tempo reale possono essere create da qualsiasi luce nella scena. Una varietà di strategie vengono utilizzate per renderle davvero performanti anche su computer datati.

Per quanto riguarda la pubblicazione di un gioco per piattaforme desktop, Unity ci dà la possibilità di usare ombre real-time per ogni sorgente di luce. Gli oggetti possono creare ombre su altri oggetti e anche su parti di loro stessi (“*self shadowing*”). Tutti i tipi di luci (*directional, spot e point*) supportano le ombre real-time che possono essere di due tipi: *Hard Shadows* oppure *Soft Shadows*.

Curiosamente le migliori ombre sono quelle non real-time. Quindi nel caso che la geometria del livello di gioco e l’illuminazione siano statici è consigliato precalcolare le lightmap nell’applicazione 3D. Le ombre calcolate in questo modo saranno di qualità migliore e più performanti piuttosto che quelle prodotte attraverso la tecnica in real-time.

Qualità delle ombre

Unity usa quelle che vengono chiamate *Shadows Maps* per creare le ombre. La tecnica dello *Shadow Mapping*³ è un’approccio basato sulle texture infatti possiamo pensare ad esse come “texture di ombre” proiettate dal punto di luce direttamente sulla scena. La qualità della mappatura dipende da due fattori:

- La risoluzione (dimensione) delle shadow map: più larga sarà la shadow map più alta sarà la qualità dell’ombra.
- Il *filtering* delle ombre: le *hard shadow* usano i pixel più vicini alla shadow map, mentre le *soft shadow* calcolano la media basata sui pixel di

³ Shadow mapping è una tecnica attraverso la quale le ombre sono state create nella grafica computerizzata in 3D. Questo concetto è stato introdotto da Lance Williams nel 1978 in un documento intitolato “Casting curved shadow on curved surfaces”. Da questo momento in poi sono state usate sia in scene pre-renderizzate che in scene real-time in diversi giochi per console e PC. In questa tecnica le ombre sono create testando se un pixel è visibile da una certa sorgente di luce ricavando il risultato da un’immagine z-buffer creata dal punto di vista di una certa sorgente di luce e che infine verranno memorizzate in forma di texture.

più shadow map ottenendo così un effetto risultante più morbido (sono più pesanti nel calcolo del render).

2.3.6 Screen Space Ambient Occlusion (SSAO)

Unity Pro presenta SSAO tra l'elenco di image effects in *post-rendering* che è possibile trovare nell'elenco dei vari componenti standard. Si può aggiungere questo componente a ogni camera per ottenere un'estetica migliore nel gioco ed è molto facile da usare. Si integra completamente con qualsiasi pipeline di rendering scelta.

La tecnica dello *Screen Space Ambient Occlusion*⁴ approssima l'*Ambient Occlusion* nel real-time come effetto d'immagine in post-processing. L'effetto che crea è di rendere più scuri pieghe, buchi e superfici che sono vicini tra di loro ed è simile a come si preserva la luce nella vita reale dove certe zone in cui la luce ambientale viene bloccata appaiono più scure alla vista.

2.3.7 Sun Shafts & Lens Effects

Unity aggiunge anche *Sun Shafts / Godrays* (luce volumetrica) come effetto interno al motore. Basta aggiungere il componente ad una Camera e ottenere un effetto immediato di luce volumetrica nell'ambiente circostante. L'effetto di immagine sun shafts simula la dispersione della luce radiale (anche conosciuto come l'effetto del "raggio di Dio") che si verifica quando una fonte di luce molto luminosa è parzialmente oscurata.

Inoltre Unity simula anche altri effetti come *Lens Flare* (bagliori di luce). L'effetto lens flares simula le luci rifrangenti all'interno dell'obiettivo della fotocamera. Viene utilizzato per rappresentare le luci molto luminose o, più in dettaglio, solo per aggiungere un pò più di atmosfera in alcune zone dell'ambiente circostante.

⁴ L'Ambient Occlusion tende ad approssimare il modo in cui la luce è radiata nella realtà ed è una tecnica di illuminazione globale. Ciò significa che l'illuminazione in ogni punto è in funzione della geometria totale nella scena. Tuttavia è una semplice approssimazione a quella è che l'illuminazione globale. L'effetto ottenuto usando questa tecnica è simile al modo in cui un oggetto potrebbe apparire in una giornata nuvolosa.

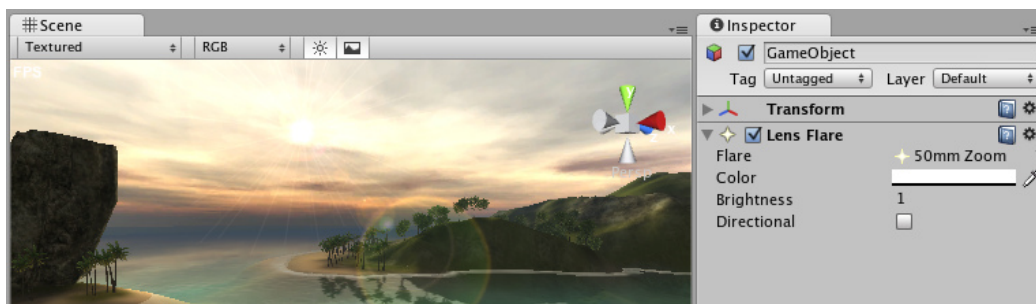


Fig. 2.18 Il pannello Lens Flare Inspector



Fig. 2.19 Esempio dell'effetto Sun Shafts

2.3.8 Lightmapping

Per avere un controllo preciso dell'illuminazione nell'ambiente del gioco l'utilizzo del lightmapping ⁵ è l'unico modo di procedere. Con Unity 3 è stato integrato uno dei migliori lightmapper esistenti, ovvero Beast. Grazie a questo strumento è possibile creare le scene di gioco direttamente all'interno di Unity aumentandone la qualità visiva.

⁵ Una lightmap è una struttura dati che contiene la luminosità delle superfici in applicazioni di grafica 3D come i videogiochi. Le lightmaps sono pre-calcolate, e utilizzate normalmente per gli oggetti statici. Sono particolarmente adatte per ambienti urbani e interni con grandi superfici planari.

Light Probes (sonde di luce)

Le *Light Probes* aggiungono realismo alle scene costruite con lightmap, senza aggiungere gli elevati costi di calcolo di luci dinamiche. L'aggiunta di light probes al sistema di illuminazione di Unity è consentita su luci pre-calcolate di personaggi e altri oggetti dinamici. Sebbene il *lightmapping* aggiunga molto al realismo di una scena, ha lo svantaggio che gli oggetti dinamici della scena sono resi meno realisticamente e il risultato potrebbe non essere corretto. Non è possibile calcolare le lightmap per oggetti in movimento in tempo reale, ma è possibile ottenere un effetto simile utilizzando le light probes. L'idea è che l'illuminazione viene campionata in punti strategici della scena, indicati con le posizioni delle sonde. L'illuminazione in qualsiasi posizione può essere approssimata con un'interpolazione tra i campioni prelevati dalle sonde più vicine. L'interpolazione è abbastanza veloce per essere utilizzata durante il gioco e consente di evitare la separazione tra l'illuminazione di oggetti in movimento e oggetti statici che fanno uso di lightmap nella scena. È necessario piazzare le sonde di luce dove si vogliono creare zone di luce o zone di ombra e formare un volume d'azione tra le varie sonde.

Dual Lightmapping

Avere degli ambienti perfettamente illuminati non è tutto. Infatti se si vuole che i personaggi principali si integrino perfettamente con le nostre lightmaps, è possibile usare il dual lightmapping che è supportato da Unity. In questa tecnica viene usata una lightmap per gli oggetti lontani mentre l'altra lightmap contiene solo la luce che viene riflessa e che rimbalza da ogni direzione.

UV Unwrap


Non è necessario fare l'unwrap dei modelli a mano a meno che non si voglia utilizzarlo per forza – infatti Unity si occuperà di farlo per noi.

Operazione di Bake

Per poter creare le lightmap si utilizza il comando di *Bake* aprendo il pannello delle lightmap e configurando le varie impostazioni. Unity quindi avvierà il processo in background consentendo di continuare a lavorare mentre è in

esecuzione. Per un controllo completo è possibile specificare una delle configurazioni messe a disposizione da Beast, in questo modo possiamo sfruttare l'intera gamma di opzioni disponibili dal lightmapper.

2.3.9 Fisica

Unity fa utilizzo del potente motore fisico NVIDIA® PhysX®  il quale permette di creare diversi tipi di comportamenti per gli oggetti grazie alle sue innumerevoli funzioni interne. Nel videogioco creato in questa tesi sono state adottate due delle caratteristiche principali offerte dal motore fisico: i *Collider* e il *Raycasting*.

I *Collider* sono primitive che definiscono i contorni degli oggetti come potrebbero essere i *Rigidbody* o i *Character Controller* e che permettono di creare le collisioni. Infatti per simulare quest'ultime tra gli oggetti nel gioco sono stati usati i *Collider* combinati all'utilizzo della funzione interna di Unity `OnTriggerEnter()`. In questo modo ogni volta che il personaggio entra in contatto col *Collider* di un ostacolo o di un oggetto viene chiamata la funzione `OnTriggerEnter()` che si occuperà di gestire poi il rallentamento della scena e tutti i comportamenti successivi.

Attraverso l'uso del *Raycasting* si crea un raggio che parte dall'oggetto e va a scontrarsi contro i *Collider* della scena. Il *Raycasting*, quindi, è stato usato per la creazione dello script inerente al posizionamento di ostacoli a random sulla scena. Questo perchè, essendo i moduli di forma circolare, gli ostacoli creati dovranno essere allineati alla normale alla superficie in cui verranno posizionati. Per fare questo l'algoritmo di randomizzazione si occuperà di allineare il vettore corrispondente all'asse z dell'oggetto in locale alla normale alla superficie. In questo modo avremo il posizionamento e la rotazione esatta dell'ostacolo sul modulo inerente.

PlayMaker

PlayMaker è un potente editor visuale di *State Machine* e una *Runtime Library* per Unity3D. Playmaker rende facile ed intuitivo l'utilizzo delle *Finished State Machine* (FSM). Nella programmazione una runtime library è uno speciale insieme di librerie usate dal compilatore, per implementare le funzioni del linguaggio di programmazione durante l'esecuzione (runtime) di un programma. Nell'utilizzare le FSM si avrà il modo di apprezzare anche l'editor grafico di Playmaker e gli strumenti di debug interni [7].

3.1 Installazione

Si consiglia di fare sempre un backup dei progetti prima di importare un aggiornamento. Per quanto riguarda i requisiti di installazione PlayMaker richiede l'uso di Unity 3.4 o superiore. Per installare PlayMaker attraverso l'estensione "unitypackage" questi sono i passi da seguire: scegliere dal menù principale di *Unity Assets > Import Package > Custom Package*. Localizzare la posizione del file *Playmaker.unitypackage* ed aprirlo. Per importare anche le scene di esempio, importare *il PlayMakerSamples.unitypackage* incluso.

NOTA: Non importare PlayMakerSamples in un progetto esistente! In caso contrario, si potrebbero sovrascrivere le risorse!

Una volta installato Playmaker all'interno di Unity verrà aggiunto un menù relativo a Playmaker accanto alle finestre degli altri menù.

3.2 Concetti base sulla State Machine

Un videogioco è paragonabile ad un automa a stati finiti poichè la risposta ad un input dipende in maniera diretta da uno stato o configurazione che comprende tutto ciò che è presente nell'ambiente di gioco, dalla scena attualmente visibile allo stato dei personaggi (e tanto altro). L'output (uscita) prodotto dal videogioco dipende sia dallo stato (scena del videogame e stato del personaggio) che dagli ingressi (input). Infatti nella definizione del modello di un automa a stati finiti il funzionamento prevede che, quando l'automa riceve un segnale di ingresso, effettui dapprima la transizione verso lo stato futuro e poi calcoli l'uscita in funzione del nuovo stato acquisito.

Per quanto detto nell'introduzione, è possibile definire automa un qualunque dispositivo in grado di eseguire una sequenza di azioni / operazioni (istruzioni), in modo automatico e senza l'intervento manuale di una persona. In particolare, l'automa eseguirà le operazioni nell'ordine stabilito per produrre i risultati attesi.

L'automa è dotato di meccanismi che gli consentono di acquisire elementi dall'esterno e produrre elementi verso l'esterno ed è un sistema che può trovarsi in diverse configurazioni più o meno complesse, che può assumere un insieme finito di stati, e che evolve in base agli stimoli od ordini ricevuti in ingresso.

Una macchina a stati finiti organizza i comportamenti in stati discreti: On, Off, Aperto, Chiuso, Camminare, Inattivo, Attacco, Difesa ... [8]

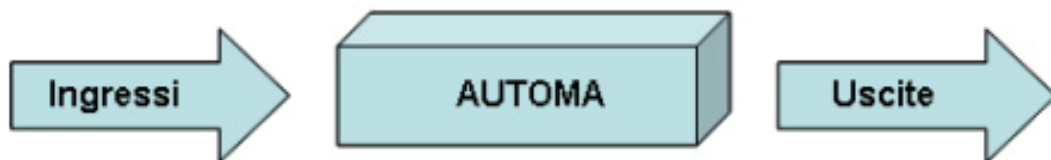


Fig. 3.1 Schema di un Automa a Stati Finiti

Playmaker utilizza le FSM per controllare i *GameObject* di Unity e rappresenta le FSM in modo visuale.

Ecco un esempio di FSM nella figura sottostante:

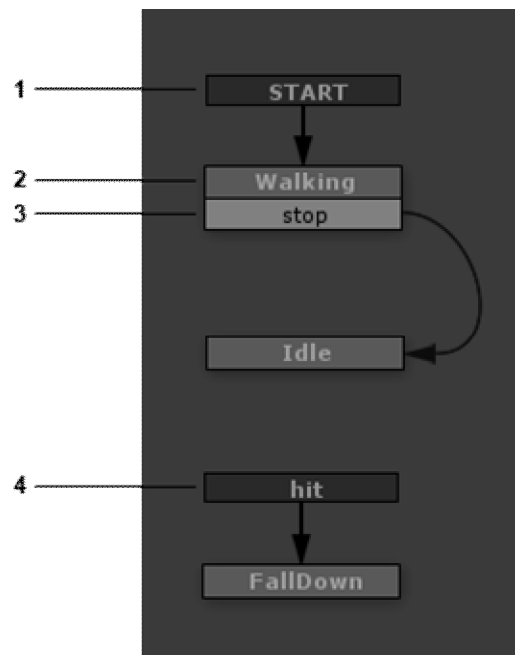


Fig. 3.2 Esempio di State Machine

1. Start Event (Evento di Partenza)

- Uno Start Event viene lanciato quando la FSM viene attivata
- Lo Start Event attiva il primo stato, conosciuto come Start State

2. Stato

- Può essere attivo solo uno stato alla volta
- Lo stato attualmente attivo esegue Azioni che controllano i GameObject

3. Transizioni

- Gli stati sono connessi da transizioni
- Il rettangolo mostra l'evento che lancerà la transizione (per esempio: *stop*, *activate*, *attack*...)
- La freccia mostra il collegamento dall'evento allo stato che attiverà

- Gli eventi possono derivare da un gran numero di casi: collisioni, *triggers* (attivatori), input utente, misurazioni della distanza, timeouts, scripts...

4. Transizioni Globali

- Una transizione globale può essere lanciata in qualsiasi momento nell'esempio in figura infatti il personaggio può essere colpito e cadere a terra in qualsiasi momento
- Le transizioni globali possono essere usate per semplificare le FSM riducendo il numero di transizioni che abbiamo bisogno di definire

3.3 Eventi

3.3.1 Introduzione

Tutte le transizioni da uno stato all'altro sono lanciate da Eventi. Useremo l'*Event Manager* per aggiungere o modificare Eventi. Ci sono 2 tipi di Eventi base:

- *System Events*: lanciati automaticamente; non possono essere modificati o cancellati
- *User Events*: eventi creati dall'utente

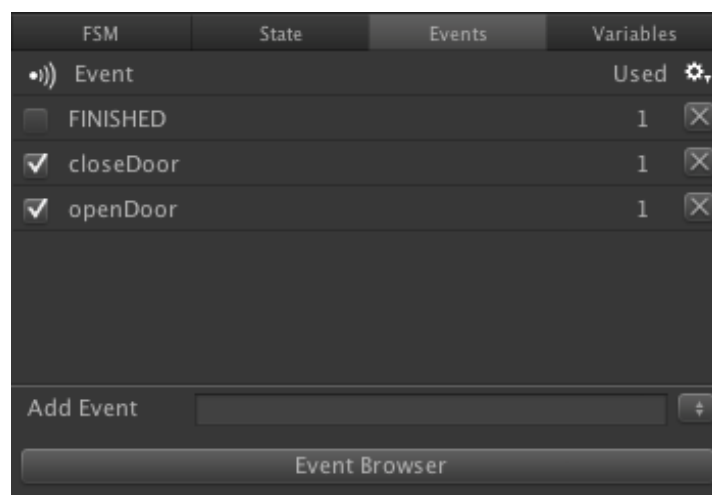


Fig. 3.3 L'event manager ci permette di editare gli eventi di una FSM.

3.3.2 Eventi di sistema (per convenzione in maiuscolo)

Evento	Descrizione	Unity Docs
START	Sent when the state machine is enabled.	N.A.
FINISHED	Sent when every action on the active state has finished.	N.A.
LEVEL LOADED	Sent after a new level is loaded.	OnLevelWasLoaded
BECAME VISIBLE	Sent when the game object becomes visible by any camera.	OnBecameVisible
BECAME INVISIBLE	Sent when the game object is no visible by any camera.	OnBecameInvisible
COLLISION ENTER	Sent when the game object first collides with another object.	OnCollisionEnter
COLLISION EXIT	Sent when the game object stops colliding with another object.	OnCollisionExit
COLLISION STAY	Sent while the game object is colliding with another object.	OnCollisionStay
MOUSE DOWN	Sent when the mouse clicks on the game object.	OnMouseDown
MOUSE DRAG	Sent while the mouse button is down and over.	OnMouseDrag
MOUSE ENTER	Sent when the mouse rolls over the game object.	OnMouseEnter
MOUSE EXIT	Sent when the mouse rolls off the game object.	OnMouseExit
MOUSE OVER	Sent while the mouse is over the game object.	OnMouseOver
MOUSE UP	Sent when the mouse button is released.	OnMouseUp
TRIGGER ENTER	Sent when the game object enters a trigger volume.	OnTriggerEnter
TRIGGER EXIT	Sent when the game object exits a trigger volume.	OnTriggerExit
TRIGGER STAY	Sent while the game object stays inside a trigger volume.	OnTriggerStay
CONTROLLER COLLIDER HIT	Sent when the character controller collides.	OnContollerColliderHit

3.3.3 Eventi Utente

É possibile aggiungere l'evento desiderato nell'*Event Manager* per controllare la propria State Machine creando un nuovo evento ed assegnandogli il nome voluto.

Per poter lanciare eventi alla FSM esistono le seguenti modalità:

1. Usando Azioni
2. Usando Eventi di animazione
3. Lanciando Eventi dagli script

Le azioni di base per lanciare eventi sono le seguenti:

- *Send Event* (lancia l'evento specifico dopo un delay a scelta)
- *Send Event To FSM* (lancia l'evento specifico ad un'altra FSM dopo un ritardo a scelta)

Altre azioni lanciano eventi basati su condizioni:

- *Collision Event*: lancia l'evento di collisione specifico quando il proprietario collide con un oggetto che ha come proprietà un tag; il collider deve includere un *Rigidbody* per poter usare la Fisica dentro Unity; attraverso i *Rigidbody* si permette all'oggetto di essere controllato dalla Fisica, mentre i *collider* permettono all'oggetto di collidere con un altro oggetto; i *collider* devono essere aggiunti agli oggetti indipendentemente dai *Rigidbody* per poter avere una collisione.
- *Trigger Event*: lancia un evento quando l'oggetto collide con un *trigger*; per usare i *trigger* è necessario marcare il *collider* come *trigger* inoltre per lanciare eventi quando due *trigger* collidono tra di loro bisogna includere un *Rigidbody* o un *Character Controller* mentre per far collidere un *trigger* con un *collider* normale uno di loro deve avere un *Rigidbody*; il *trigger* è un oggetto bidimensionale o tridimensionale invisibile al *player* il quale quando verrà a contatto con un oggetto invierà un evento tramite cui è possibile lanciare azioni o comportamenti.

- *Mouse Pick Event*: lancia eventi basati sull'interazione del mouse con un oggetto.

Ci sono un gran numero di azioni documentate; per poterle consultarle accedere alla documentazione online a questo indirizzo:

<https://hutonggames.fogbugz.com/default.asp?W2>

Eventi lanciati da Animazioni:

Possiamo usare la finestra Animation di Unity per lanciare eventi alle FSM.

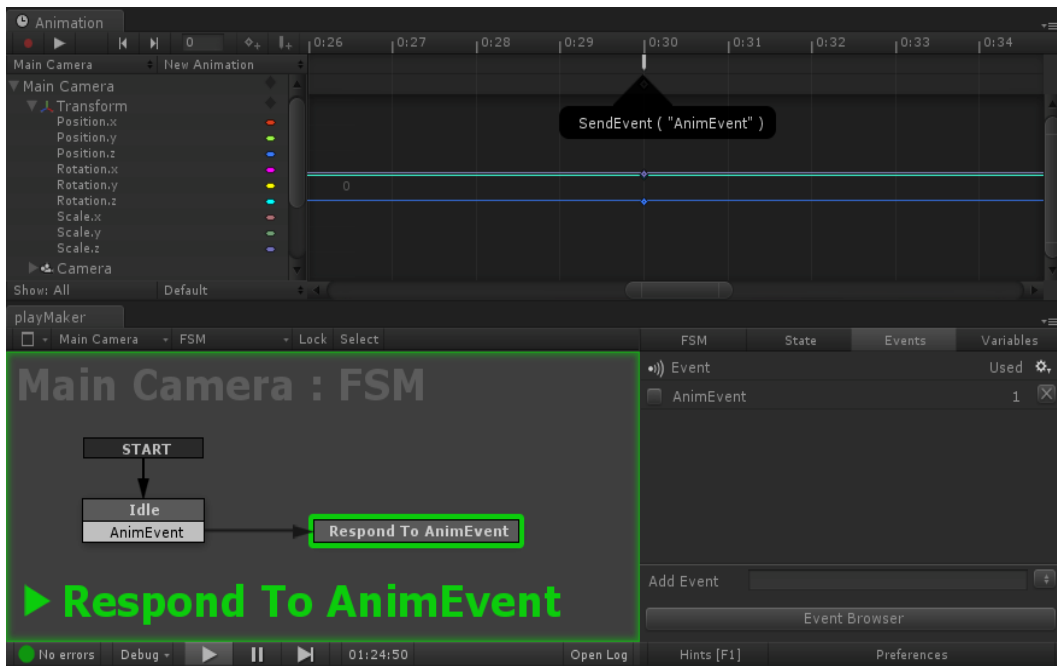


Fig. 3.4 Finestra Animation interna a Unity

3.4 Azioni

3.4.1 Introduzione alle Azioni

Ogni stato ha una lista di azioni da eseguire quando è attivo. Ogni azione ha uno specifico scopo: far partire una animazione, far muovere un oggetto verso un target, fare un test per una collisione etc. Semplici azioni possono essere combinate per creare comportamenti complessi.

3.4.2 Ordine di esecuzione

Le azioni sono attivate nell'ordine in cui sono listate, dall'alto verso il basso. È possibile spostarle in alto/basso usando *l'Action Editor*. Se un'azione lancia una transizione, la transizione parte immediatamente e le azioni sottostanti non vengono eseguite.

Infine un'azione non deve per forza finire prima che la prossima azione venga eseguita - l'intera lista di azioni viene valutata frame per frame. Questo significa che azioni multiple possono essere anche simultanee. (per esempio le azioni: *Move Towards* e *Look At*).

3.4.3 Ciclo di vita di una Azione

Le azioni hanno un ciclo di vita: quando sono attive partono con l'esecuzione e poi si aggiornano finché terminano o finché lo stato attuale non è terminato. Per ogni frame tutte le azioni nella lista che non sono terminate si aggiornano partendo dall'alto verso il basso. Le azioni sono colorate di verde per indicare quando sono attive.

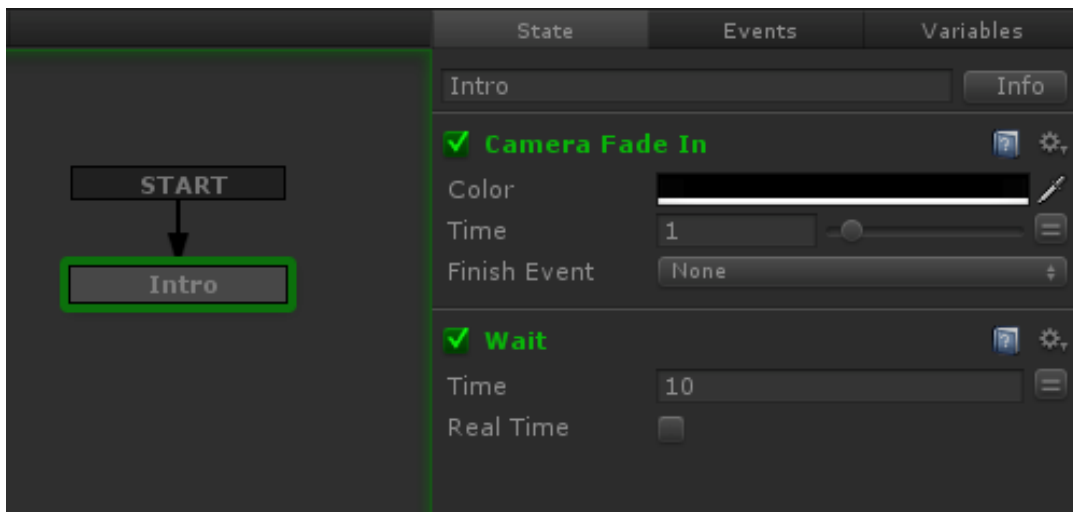


Fig. 3.5 Dopo aver premuto Play lo stato Intro diventa attivo e tutte le sue azioni vengono attivate.

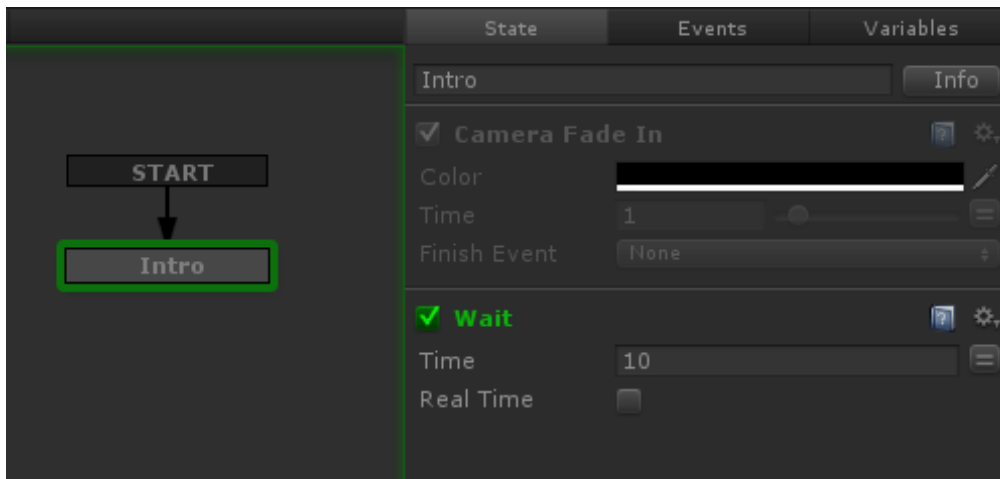


Fig. 3.7 Dopo un secondo l'azione Camera Fade è completata. L'azione si colora di grigio per mostrare che è finita.

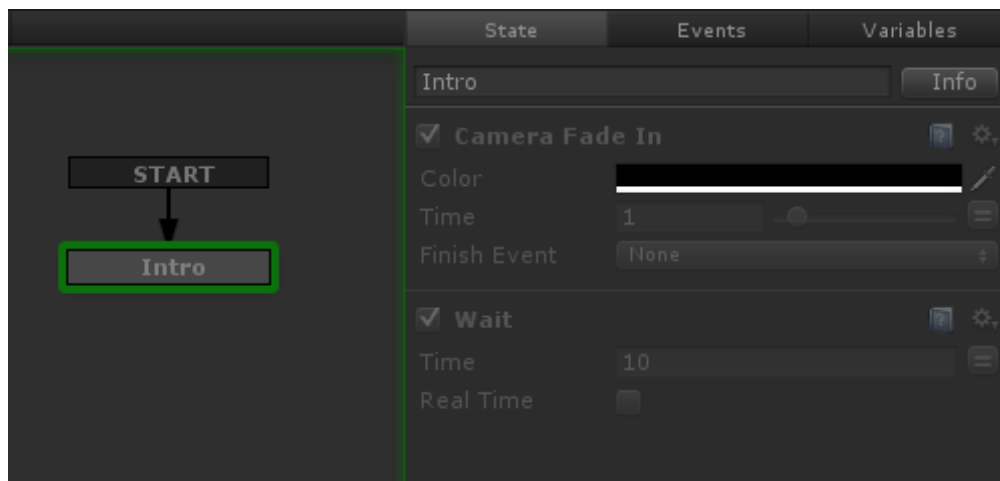


Fig. 3.8 Dieci secondi dopo che lo stato è stato attivato, l'azione di Wait è completa e scompare.

È importante ricordare che le azioni vengono eseguite contemporaneamente, quindi l'azione di *Wait* sarà eseguita per l'intera durata. Azioni che hanno bisogno di essere avviate in sequenza devono essere in stati separati collegati da transizioni. Ogni azione ha il suo criterio di terminazione. Il criterio dipende dal tipo di azione: alcune azioni finiscono istantaneamente come ad esempio *Set Material*, *Stop Animation*, *Enable Behaviour*; altre azioni hanno bisogno di tempo per terminare come ad esempio *Move Towards*, *Camera Fade In*, *Float Interpolate*; altre azioni ancora non finiscono mai e sono progettate per rimanere attive per tutto il tempo in cui uno stato è attivo, vedi ad esempio *Blink*, *Flicker*, *Mouse Pick Event*.

Molte azioni hanno una opzione *Every Frame* che per ogni frame indica a loro di ripetersi:

- Se *Every Frame* è impostato a *True*: l'azione si ripete per ogni frame e va avanti finchè lo stato non è più attivo.
- Se *Every Frame* è impostato a *False*: l'azione viene eseguita una volta poi finisce.

Quando tutte le azioni di uno stato sono terminate, un evento *FINISHED* viene lanciato alla State Machine. Possiamo usare questo evento per avere una transizione ad un nuovo stato.

3.5 Concetti base sull'Editing

Abbiamo visto alcune nozioni riguardanti i concetti fondamentali di Playmaker ora questo capitolo vuole mettere in relazione i punti spiegati e descrivere il flusso di lavoro per poter usare l'editor.

3.5.1 Avviare Playmaker

Dopo aver installato il tool aprire il Playmaker Editor dal menu di Playmaker:

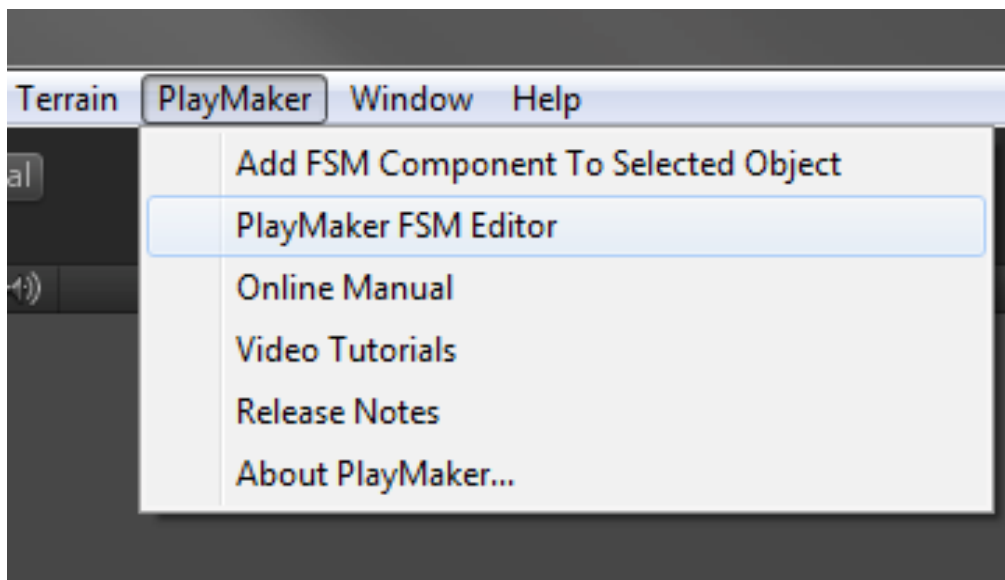
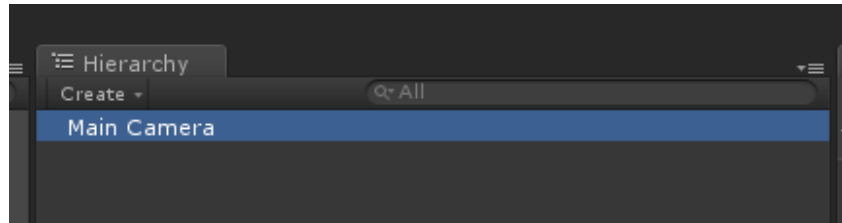


Fig. 3.9 Menu di PlayMaker

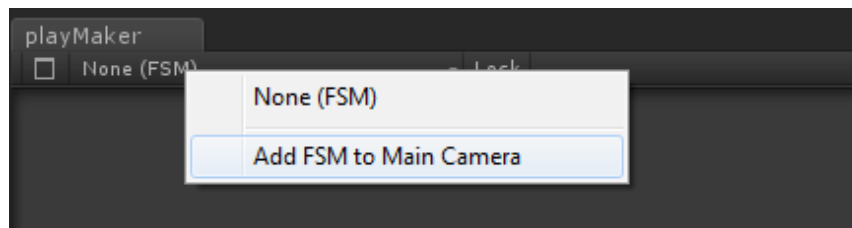
3.5.2 Aggiungere una FSM ad un Game Object

Metodo 1

1. Selezionare il GameObject:



2. Aprire il selettore di FSM dalla *Toolbar* di selezione:



3. Selezionare "Add FSM to [game object name]"

Ora il GameObject ha una componente FSM di Playmaker ed è automaticamente selezionata nell'editor.

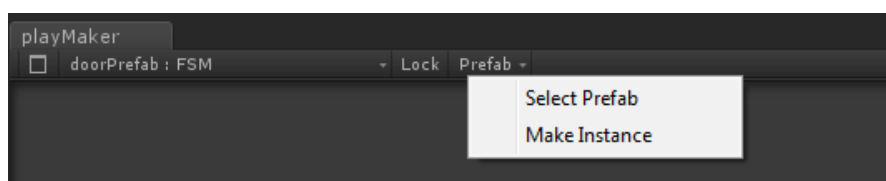
Metodo 2

Selezionare il *GameObject*; selezionare dal menu: *Playmaker* > *Add FSM Component*. Il *GameObject* ora ha una componente FSM di Playmaker che è automaticamente selezionata nell'editor.

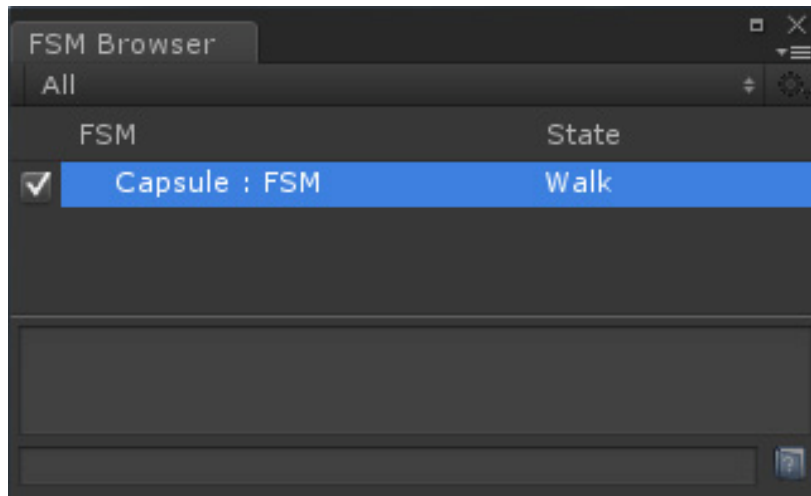
3.5.3 Selezionare una FSM

Ci sono diversi modi per selezionare una FSM. Si può semplicemente selezionare un *GameObject* nel progetto. Se l'editor di Playmaker è aperto la FSM viene automaticamente selezionata.

Oppure usando la *Toolbar* di Selezione:



Usando il *Browser* per le FSM:



Selezionare un *GameObject* e individuare il suo componente FSM nel Pannello di Ispezione. Cliccare il tasto *Edit* per aprire l'editor di Playmaker e selezionare la FSM.

Aggiungendo una FSM al *GameObject* e usando la *Toolbar* di Selezione la FSM verrà automaticamente selezionata.

3.5.4 Selezionare gli Stati

Ci sono due modi di selezionare stati: cliccando e trascinando gli stati selezionati nel *Graph View* oppure usando lo *State Browser*.

Quando il gioco è in play, gli stati attivi sono automaticamente selezionati in modo tale che si possano vedere le loro azioni. È possibile disattivare questa caratteristica entrando nelle Preferenze.

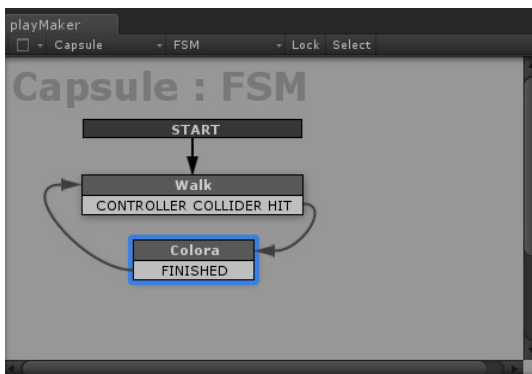


Fig. 3.10 Il Graph View di PlayMaker.

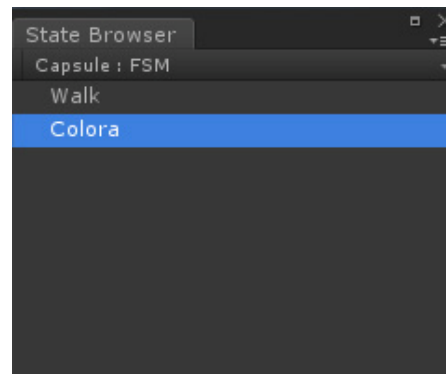


Fig. 3.11 Lo State Browser di PlayMaker.

3.5.5 Impostare lo Stato di Start

Lo stato di *Start* è lo stato lanciato quando la FSM è attiva e viene indicato da una transizione *START*. Si può impostare lo Stato di Start cliccando col tasto destro sullo stato desiderato e selezionando *Set As Start State*.

3.5.6 Aggiungere Transizioni tra Stati

Le transizioni tra stati sono lanciate da Eventi, quindi dobbiamo aggiungere qualche evento tramite *l'Event Manager*. Cliccando col tasto destro su uno stato selezionare *Add Transition*. Questo aggiunge una nuova transizione che posso modificare.

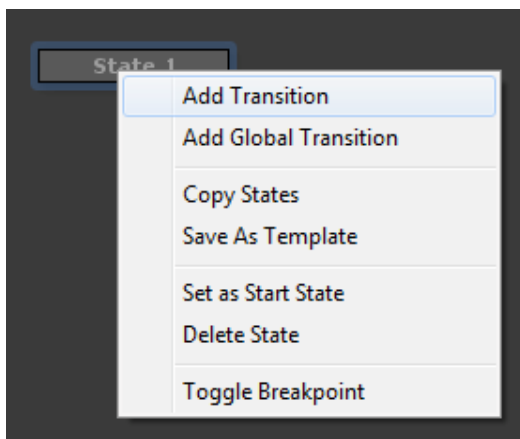


Fig. 3.12 Aggiunta di una transizione.

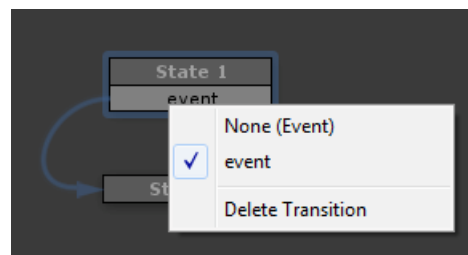


Fig. 3.13 Collegamento tra uno stato e l'altro.

Ora trascinare un collegamento dall'evento allo stato in cui si vuole fare arrivare la transizione. Ripetere tutto il processo finchè tutti gli stati sono connessi come si desidera.

3.5.7 Aggiungere Azioni ad uno Stato

Selezionare lo Stato a cui aggiungere l'azione e poi aprire *l'Action Browser* dal *Main Menu* o dallo *State Inspector*. Selezionare l'azione e cliccare su *Add Action*. L'azione scelta apparirà nello *State Inspector*.

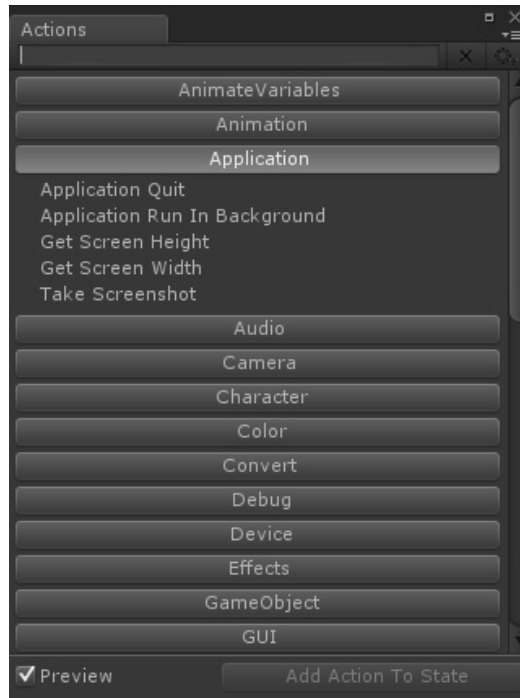


Fig. 3.13 L'Action Browser di PlayMaker.

3.5.8 Variabili di State Machine

Le *State Machine* possono usare variabili per creare comportamenti più complessi. Si possono creare e modificare variabili nel *Variabile Manager*. Una variabile in PlayMaker è un contenitore indentificato da un nome e contenente un valore. Molte azioni sfruttano diversi vantaggi prendendo in ingresso variabili codificate come valori all'interno del programma.

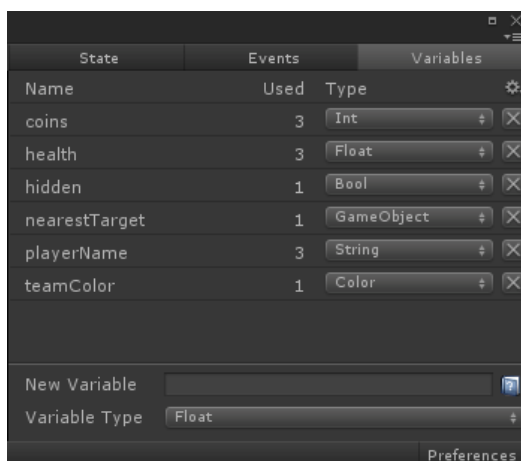


Fig. 3.14 Il Variable Manager di PlayMaker.

Capitolo 4

Modellazione

In questo capitolo viene preso in esame il software scelto per la modellazione degli oggetti e delle scene del videogioco: 3D Studio Max. Possiamo trovare un accenno ai concetti di base per l'utilizzo ed all'interfaccia grafica del programma. Inoltre vengono analizzate le tecniche di modellazione più importanti per la creazione di alcuni modelli usati. Grazie alla computer grafica è stato possibile dare vita alla scena precedentemente visualizzata attraverso lo *storyboard*. Tra le tecniche più importanti vi sono: la modellazione, l'ossatura, la creazione della pelle ed l'animazione del modello. Viene spiegato come esportare il modello creato tramite 3D Studio Max e come importarlo in Unity. Infine vengono analizzate le tecniche per il *texturing* e l'illuminazione dei modelli.

4.1 3D Studio Max

4.1.1 Introduzione

3D Studio Max è un potente programma che permette la realizzazione di immagini, animazioni ed effetti speciali in computergrafica. È uno strumento che si rivela molto utile in vari campi (dall'architettura alla realizzazioni di effetti speciali od anche alla moda) soprattutto grazie all'innumerabile numero di funzioni che ingloba ed alla grande quantità, di plug-in esterni che permettono di realizzare i progetti più disparati in maniera sempre più veloce ed accurata, come ad esempio per realizzare terreni, stoffe e tessuti, o per simulare addirittura i peli ed i capelli per modelli che rappresentano animali o umanoidi [5].

Si tratta di un programma object-oriented: ogni oggetto che si crea nella scena (modelli, luci o telecamere) possiede un nome, delle proprietà, delle caratteristiche ed una memoria riguardante le modifiche che gli sono state applicate. Questi

attributi possono essere manipolati in ogni istante permettendo di ripercorrere e modificare la storia dell'oggetto offrendo così all'artista una grande flessibilità durante la realizzazione della scena. Un'altra fondamentale caratteristica è la possibilità di realizzare le animazioni in real-time variando le posizioni, le rotazioni od il *morphing* degli oggetti nello spazio ed in relazione al tempo in maniera visuali, senza dovere per forza definire il tutto tramite *keyframing*. Dopo questa breve panoramica sulle caratteristiche del pacchetto vedremo le principali caratteristiche.

4.1.2 Interfaccia: Viste e Strumenti Principali

Aperto 3D Studio Max verranno mostrate quattro tipi di Viste, ognuna con la propria griglia di riferimento. Cliccando col tasto del mouse sul nome di una vista è possibile sceglierne un'altra tra quelle disponibili. Ad esempio se si vuole sostituire la vista *Left* con la vista *Right*, è necessario cliccare sul nome *Left* della vista e impostare *Right*.

Oltre all'area delle viste, esistono altre quattro aree principali. In alto troviamo la Barra degli Strumenti principali, a destra il *Pannello dei Comandi*, in basso a destra i comandi di Gestione delle Viste e sempre in basso la Barra Temporale e i comandi relativi alla riproduzione dell'animazione nelle viste.

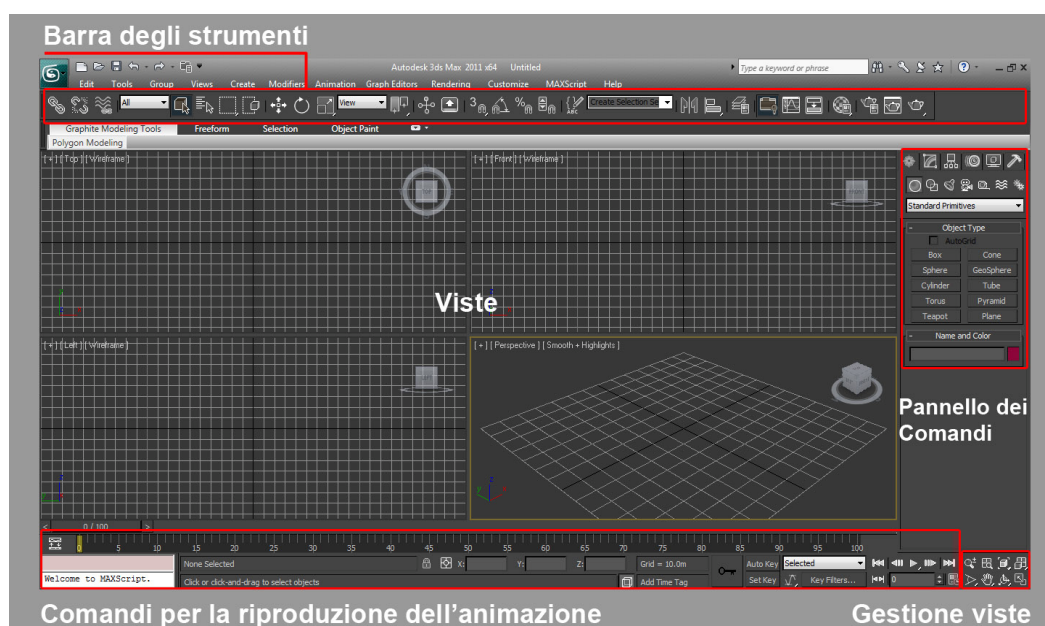


Fig. 4.1 Interfaccia principale di 3D Studio Max

Andando più in particolare all'interno della Barra degli Strumenti si trovano i tasti presenti nella figura 4.2 i quali servono per traslare, ruotare e riscaldare l'oggetto selezionato. Cliccando col tasto destro del mouse su ognuno dei tre tasti si aprirà una schermata che permetterà l'inserimento dei valori riferiti agli assi cartesiani (x,y,z). I tasti di figura 4.3 servono rispettivamente per attivare il modulo di realizzazione dei materiali che, poi, servirà per ricoprire gli oggetti che compongono la scena e per attivare la renderizzazione; il processo di calcolo per ottenere la resa videorealistica della scena creata.



Fig. 4.2 Trasformazioni



Fig. 4.3 Material Editor / Render

Il Pannello dei Comandi rappresenta il centro nevralgico del pacchetto, infatti da questo menù si possono creare, modificare, collegare gerarchicamente ed animare gli oggetti che popolano la scena (ed inoltre accedere ai vari plug-in esterni). Muovendosi tra le varie funzioni questo menu si espanderà verso il basso e quindi si potrà fare scorrere il menu in verticale. Il Pannello dei Comandi è suddiviso in sei parti:

- *Create*: permette di creare gli oggetti utilizzando varie tecniche (partendo dalle primitive, per estrusione, patch modelling, ecc). Questo può essere suddiviso a sua volta in sette parti (geometrie, forme, luci, camera, helpers, space warps, systems). Nella fig. 4 si possono vedere alcune delle geometrie possibili (Nota: ognuna delle scelte farà apparire un sottomenu che contiene tutti i parametri che la caratterizzano.)
- *Modify*: questa parte contiene tutta una serie di modificatori che possono agire sulle geometrie che compongono gli oggetti a vari livelli, dalle singole facce ai vertici. Questi modificatori servono per semplificare un

modello diminuendone la complessità, per variare le coordinate di mappatura, per ottenere oggetti 3d partendo da figure 2d (dette shape).

- *Hierarchy*: permette realizzare e di variare i parametri dei collegamenti tra gli oggetti, al fine di creare dipendenze e condizionamenti nelle animazioni degli oggetti.
- *Motion*: consente di avere accesso ai valori (parametri) associati alle geometrie, quali rotazioni, movimenti e variazioni di scala; inoltre permette anche all'artista di poter agire numericamente sulle traiettorie che gli oggetti seguiranno durante l'animazione.
- *Display*: contiene tutti i parametri che regolano la modalità di raffigurazione degli oggetti nelle finestre di lavoro.
- *Utilities*: racchiude i pulsanti di chiamata dei file eseguibili dei vari plug-in standard ed esterni. I pulsanti di layout possono sostanzialmente essere suddivisi in due gruppi: pulsanti per la gestione delle animazioni: avanzare, riavvolgere, eseguire ed aggiungere chiavi; gruppo di pulsanti che permettono di aggiustare la vista (dei singoli oggetti o dell'intera scena) all'interno delle viewport.

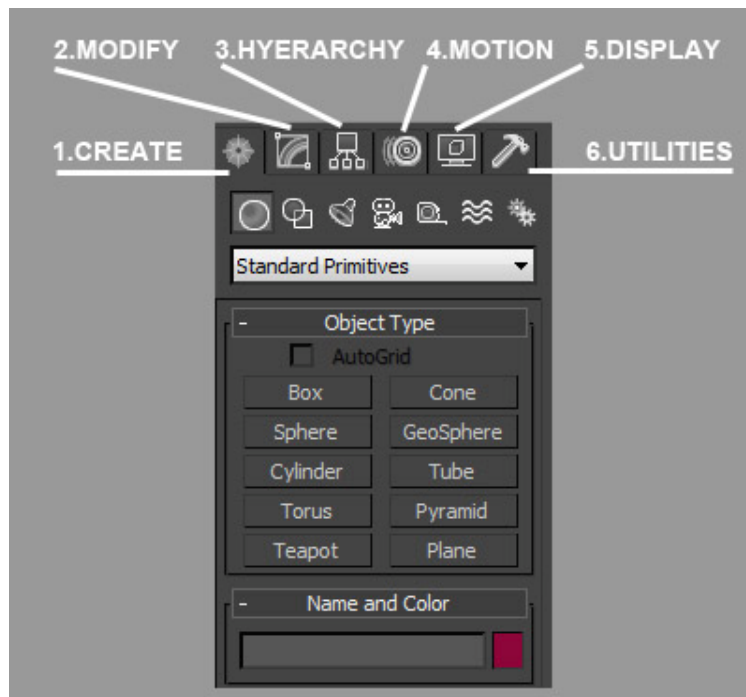


Fig. 4.4 Pannello dei Comandi

4.1.3 Le Primitive 3D Standard

Le "primitive 3D" sono i solidi geometrici predefiniti in 3D Studio Max. Le primitive standard sono quelle di uso comune come la sfera, il cono, il cilindro e altre. Per crearle basta cliccare sul pulsante relativo e trascinare la primitiva nella vista. A seconda del tipo di oggetti occorrerà trascinare e cliccare più volte per dare forma alla figura voluta. Attraverso *Create* si ottengono le figure sottostanti:

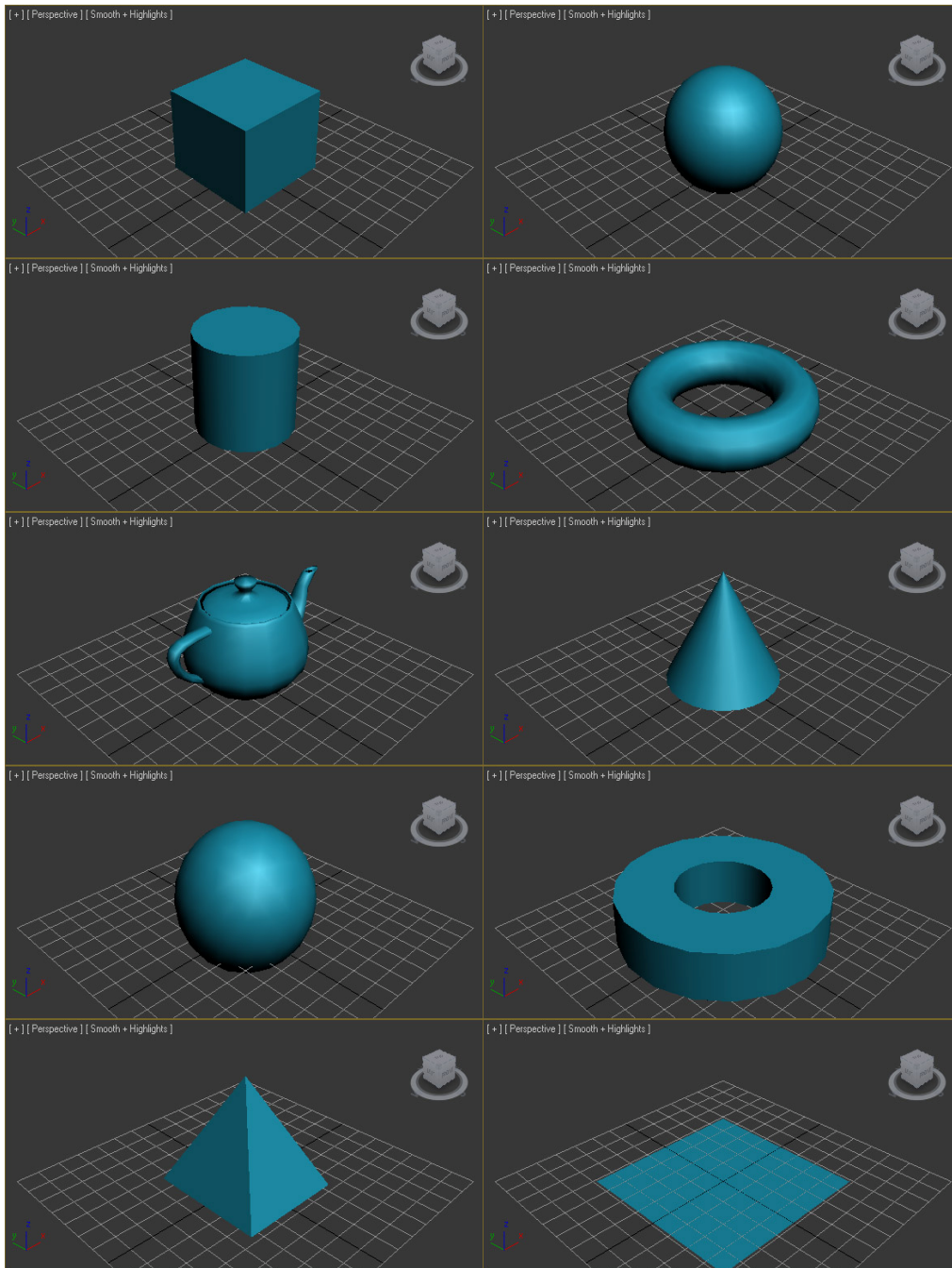


Fig. 4.5 Primitive Standard di 3D Studio Max

Per ogni primitiva 3D esiste una serie di parametri che permettono di modificarle, visualizzabili nella cartella *Modify* dopo aver selezionato la primitiva. In questo pannello, in alto è possibile definire il nome dell'oggetto e grazie al quadro colorato a destra è possibile modificare il suo colore.

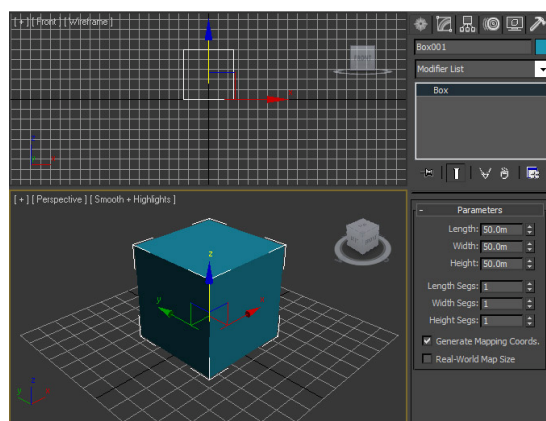


Fig. 4.6 Parametri della primitiva.

La modifica della forma di un oggetto dipende non solo dalla variazione numerica dei suoi parametri più intuitivi, come ad esempio potrebbe essere il *Radius* di una sfera, ma anche dalla variazione del valore del parametro *Segments*. Questo parametro regola la quantità di segmenti utilizzati per interpolare la superficie dell'oggetto. In generale, un oggetto con una quantità maggiore di poligoni formanti la sua superficie avrà una forma più smussata. Quello che chiede in cambio è una quantità maggiore di tempo per essere renderizzato, e una quantità maggiore di memoria RAM della scheda video per essere gestito nella vista. Proprio per questo vedremo che nella modellazione orientata ai videogiochi si cercherà di usare meno poligoni possibile per avere un minor sovraccarico delle risorse disponibili ed una scena di gioco più fluida possibile.

4.1.4 Trasformazioni Base: Traslazione, Rotazione e Scalatura

I comandi di traslazione, rotazione e scalatura si trovano in alto nella Barra degli Strumenti principali:



Fig. 4.7 Trasformazioni Base

Premendo il primo pulsante si attiverà il comando di Traslazione sulla figura scelta e compariranno delle frecce orientate secondo gli assi del sistema di riferimento che si utilizza. Trascinare un asse per spostare la figura lungo quell'asse mentre per spostare la figura sul piano individuato da due assi, trascinare il riquadro formato dai due assi.

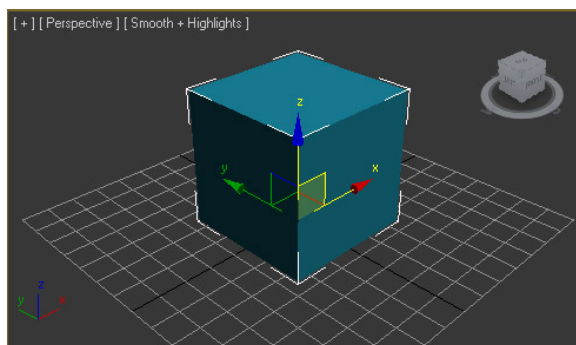


Fig. 4.8 Traslazione di una primitiva.

Premendo il secondo comando si attiverà la rotazione e compariranno delle circonferenze attorno all'oggetto. Ogni circonferenza è ortogonale a un asse del sistema di riferimento. Trascina una circonferenza per ruotare la figura intorno all'asse cui è ortogonale. Per ruotare l'oggetto contemporaneamente rispetto a tutti e 3 gli assi, trascinare la circonferenza grigia piccola. Per ruotare l'oggetto parallelamente alla vista attiva, ruotare la circonferenza grigia grande.

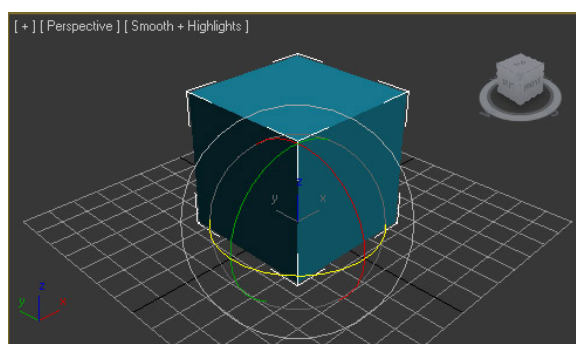


Fig. 4.9 Rotazione di una primitiva.

Infine premendo l'ultimo comando si attiverà la scalatura, e compariranno gli assi uniti da segmenti a 45 gradi. Trascinando su un asse avviene la scalatura lungo quell'asse. Trascinando sui segmenti che collegano 2 assi, la scalatura avviene contemporaneamente su quei due assi. Trascinando sul triangolo centrale, si

ottiene una scalatura uniforme dell'oggetto, cioè su tutti e 3 gli assi contemporaneamente.

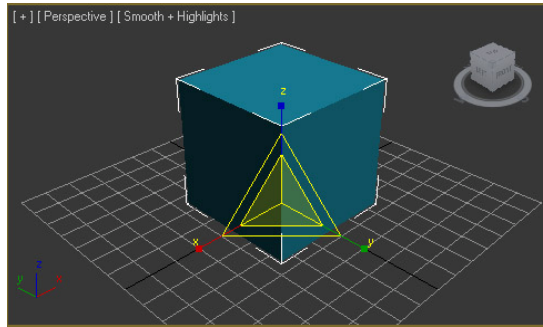


Fig. 4.10 Scalatura di una primitiva.

4.1.5 Griglie e Unità di Misura

Per regolare la griglia occorre effettuare 3 passaggi: definizione dell'unità di misura, definizione della visualizzazione dell'unità di misura e dimensionamento della griglia.

Definizione dell'unità di misura:

Tutte le dimensioni degli oggetti e la loro collocazione nello spazio sono misurati con un certo tipo di unità, chiamate unità di Max. È possibile decidere a cosa far corrispondere questa unità procedendo in questo modo: cliccare su *Units Setup* dal menu *Customize* e in seguito cliccare su *System Unit Setup*, si aprirà la finestra di dialogo *System Unit Setup*. In questa finestra diciamo al programma a cosa corrisponderà una unità di Max.

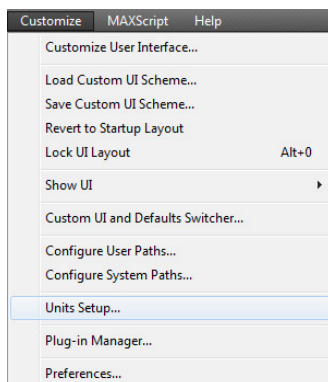


Fig. 4.11 Menu Customize > Unit Setup

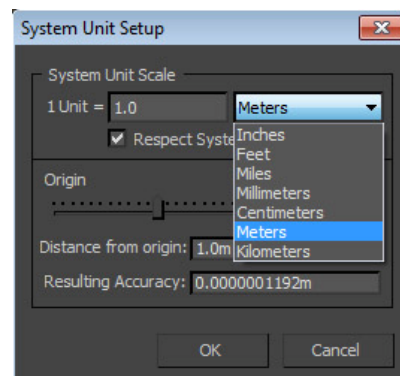


Fig. 4.12 Finestra di dialogo System Unit Setup

Definizione della visualizzazione dell'unità di misura

Una volta che si è decisa l'unità di misura da utilizzare come equivalenza per le unità di Max, cliccando su “OK” tornare alla finestra precedente. Qui si ha la possibilità di definire quale unità di misura visualizzare dopo i valori numerici, che verranno convertiti di conseguenza. Ad esempio, impostando a *Meters* la *Display Unit Scale* si otterrà che per ogni valore dei parametri di una primitiva verrà aggiunta l'unità di misura in metri.

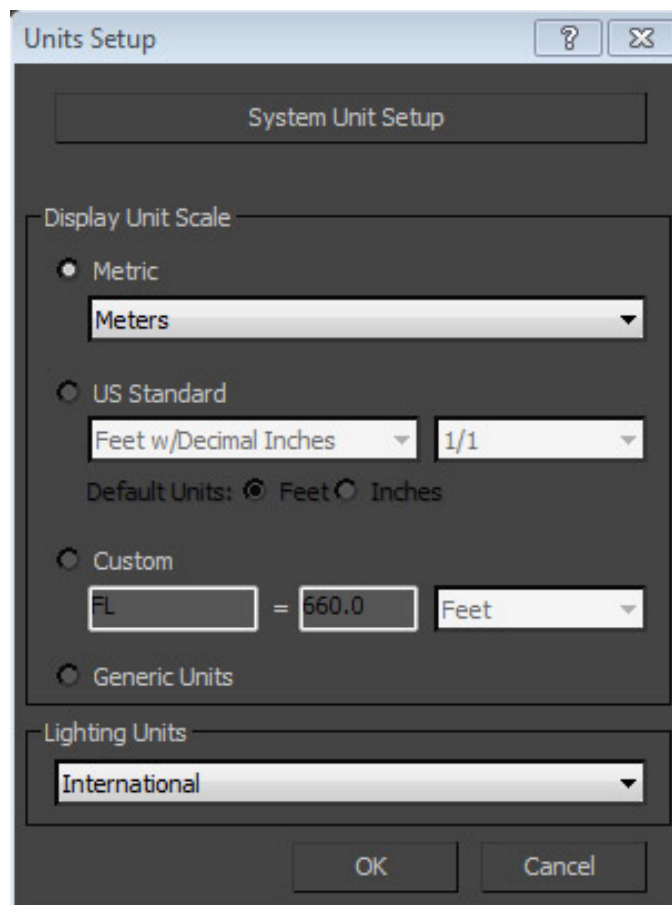


Fig. 4.13 Finestra di dialogo Units Setup

Dimensionamento della griglia

Cliccare su *Grids and Snaps* dal menu *Tools* e in seguito cliccare su *Grid and Snap Settings*, si aprirà la finestra di dialogo relativa. Nella finestra che si apre, premere sul tab *Home Grid*. In *Grid Spacing* dentro al pannello *Grid Dimensions* impostare la dimensione dei riquadri relativi alla griglia. Quindi se si vuole definire 1 riquadro = 1m, impostare 1 come valore.

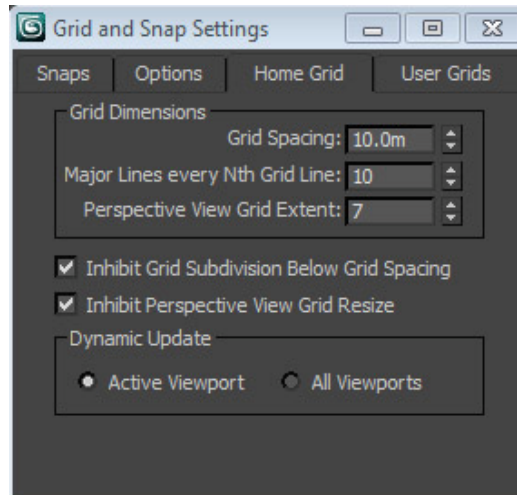


Fig. 4.14 Finestra di dialogo Grid and Snap Settings

4.1.6 Pivot Point

Il *Pivot Point* di un oggetto è il punto dal quale avviene la rotazione dell'oggetto. Ad esempio quando si applica una rotazione a una sfera, questa ruota attorno al suo centro geometrico perchè per default il suo *Pivot Point* è posizionato esattamente nel centro della primitiva. E' possibile modificare la sua posizione selezionando l'oggetto desiderato e aprendo il pannello *Hierarchy*. In seguito cliccando sul pulsante *Affect Pivot Only* tutte le trasformazioni avranno effetto solo sul *Pivot Point*, mentre l'oggetto resterà immobile.

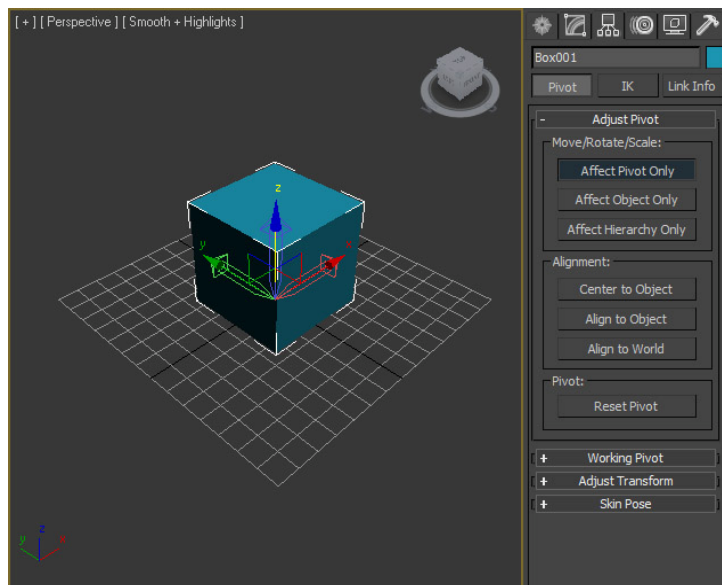


Fig. 4.15 Pannello Adjust Pivot

4.1.7 Modificatori

Gli oggetti inseriti, possono essere modificati utilizzando alcune tecniche particolari, chiamate appunto Modificatori. I più importanti sono:

- *Bend* (Piega)
- *Taper* (Rastrema)
- *Twist* (Torsione)
- *Stretch* (Allunga)
- *Wave* (Onda)

I modificatori non aggiungono nuovi vertici, ma spostano semplicemente quelli esistenti. Per poter ottenere effetti più definiti, occorre aumentare il numero di vertici da cui le primitive base sono costituite. Per molte primitive, è possibile aumentarne il numero di vertici attraverso i parametri di base che troviamo nel pannello *Create*.

I modificatori vengono inseriti esattamente come le operazioni di estrusione selezionando un oggetto ed entrando nel pannello *Modify*. È possibile impostare i parametri dei modificatori, dopo averli selezionati nell'elenco (quello selezionato viene visualizzato su sfondo scuro). È anche possibile rimuovere il modificatore selezionato, premendo l'apposito pulsante.



Fig. 4.16 Pannello Modify con lista dei Modificatori applicati

Il modificatore *Bend* (piega) piega un oggetto lungo un asse. E' possibile specificare un angolo della piega ed una direzione. La direzione specifica l'asse lungo il quale l'oggetto viene piegato.

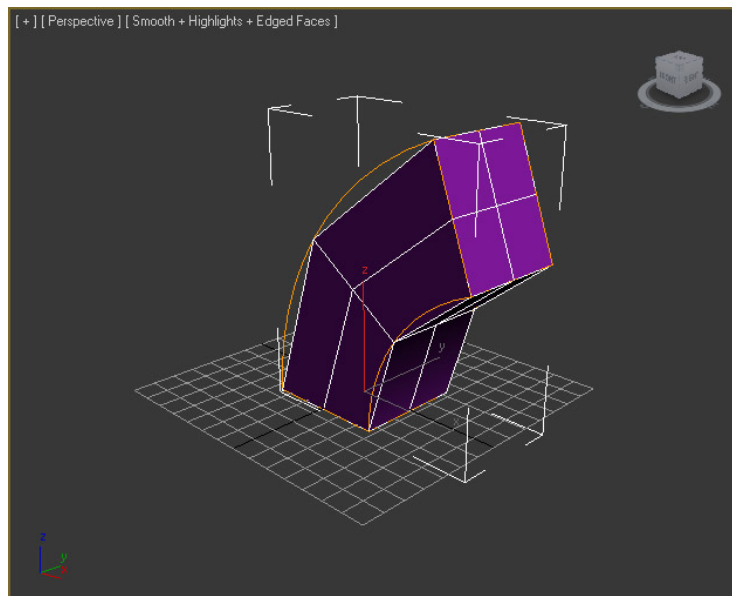


Fig. 4.17 Esempio di modificatore Bend

Il modificatore *Taper* (rastrema) schiaccia un oggetto in un tronco di piramide. I parametri sono l'ammontare dello schiacciamento ed un eventuale curvatura. Applicando una curvatura, il cuneo in cui e' inscritto l'oggetto viene arrotondato.

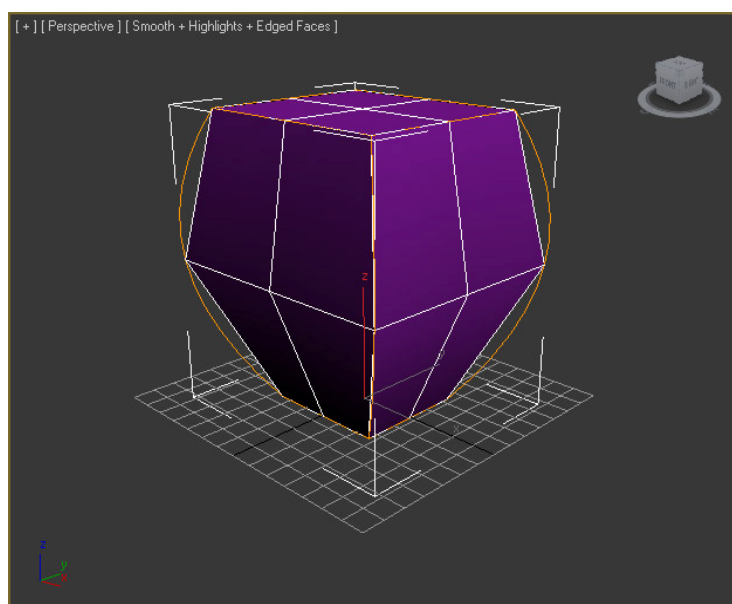


Fig. 4.18 Esempio di modificatore Taper

4.2 Creazione di un modello

4.2.1 Introduzione

La realizzazione di grafica tridimensionale e' un processo molto complicato che richiede notevole potenza di calcolo. Generare un'immagine tridimensionale puo' richiedere molto tempo. Il processo di lavorazione viene quindi suddiviso in due fasi:

- Costruzione (modellazione)
- Generazione (simulazione e rendering)

La costruzione consiste nella definizione del "mondo virtuale" che si vuole rappresentare. E' la fase che richiede piu' lavoro da parte del disegnatore, ma non impegna particolarmente il computer. La fase di costruzione e' a sua volta suddivisa in numerose attivita' distinte. Le piu' importanti sono: *Modellazione - Texturing - Illuminazione - Rendering - Rigging - Blocking - Animazione - Clothing - Dinamica - Special EFX - Compositing*. Ogniuna di esse richiede abilita' distinte e competenze specifiche. Le attivita' della fase di costruzione servono solamente a definire il mondo virtuale. Una volta che e' stato costruito, occorre "rappresentarlo".

La fase di generazione esegue le operazioni (matematiche) necessarie per creare gli elementi con i quali produrre una vista del mondo modellato. Questa fase richiede un grandissimo lavoro da parte del computer (in alcuni casi occorrono anche giorni). Il disegnatore deve solamente "aspettare" che il computer abbia finito di generare l'immagine. Molte delle attivita' svolte nella fase di costruzione (*Rendering - Clothing - Dinamica - Special EFX - Compositing*) hanno associata una fase di generazione. Normalmente questa attivita' prende il nome di rendering (nel caso di *Rendering* e *Compositing*) o di simulazione (nel caso di *Clothing*, *Dinamica* o *Special EFX*).

Nella fase di modellazione si avra' la definizione dei modelli tridimensionali che compongono una scena. Esistono numerose tecniche di modellazione. Le principali si chiamano: *poligonale*, *NURBS* e *subdivison surface*.

Il *Texturing* "dipinge" i modelli applicando immagini opportunamente disegnate od acquisite. Una componente fondamentale di tale lavoro consiste nello stabilire il modo in cui le immagini vengono applicate sui modelli 3d (definizione delle coordinate di mappatura).

Nella fase di *Illuminazione* si stabiliscono le fonti luminose che illuminano la scena. Vi sono numerosi modi in cui si puo' illuminare una scena per ottenere effetti piu' o meno realistici. L'esperto di illuminazione conosce queste tecniche e sa selezionare quella piu' opportuna.

Nel *Rendering* si stabilisce il modo in cui le texture reagiscono alle sorgenti luminose che le illuminano. Il compito di questa fase e' di fare in modo che i modelli sembrino costituiti da materiali specifici e presentino il livello di realismo desiderato. Inoltre, l'esperto di rendering sa come ottimizzare una scena per ridurre il tempo di generazione.

Una volta costruiti i modelli ed applicate loro le texture, occorre "muoverli" per creare un'animazione. Nella fase di *Rigging* (ossatura), si stabiliscono i controlli che un animatore avra' a disposizione per posizionare i modelli.

L'Animazione crea i movimenti veri e propri dei modelli tridimensionali, utilizzando i controlli creati nella fase di rigging. Vi sono diversi livelli di animazione: si parla generalmente di animazioni primarie (es. lo spostamento degli arti di un personaggio durante una camminata) e secondarie (es. le palpebre degli occhi) [1].

4.2.2 Modellazione del Ponte

Per quanto riguarda la modellazione di ogni oggetto della scena si parte da una figura di base attraverso la quale poi è possibile creare l'oggetto voluto, ad esempio partendo da un semplice box tridimensionale si arriverà a creare il modello di un ponte che sarà poi utilizzato in uno dei moduli del gioco.

Dal pannello *Standard Primitives* di 3D Studio Max selezionare un oggetto di tipo *Box*. Per l'allineamento del box sul modulo scelto è necessario utilizzare il comando attivabile attraverso la combinazione di tasti ALT+A . In questo modo è possibile allineare il box all'oggetto si cui andremo a cliccare facendo attenzione

al fatto che il riposizionamento dipende dal *Pivot* di questo ultimo. Ogni oggetto infatti ha un pivot grazie al quale si possono cambiare le coordinate di mondo o coordinate locali. Per modificare il pivot di un oggetto premere sul tab *Hierarchy* e posizionarsi nel pannello *Adjust Pivot* cliccando infine sul pulsante *Affect Pivot*. In questo modo è possibile spostare il pivot di un oggetto ovunque si voglia. Un'altra cosa che è possibile fare è centrare il pivot nell'oggetto. Cliccare sul pulsante *Center To Object*, in questo modo il programma posizionerà il pivot nel centro dell'oggetto. Ora rizelezionare il box, cliccare la combinazione dei tasti ALT+A, cliccare sull'oggetto a cui allinearlo e infine l'oggetto sarà posizionato nel centro del modulo in cui si vuole posizionarlo. Inoltre all'interno del pannello align selection si può selezionare l'orientamento locale sui tre assi per impostare l'orientamento della primitiva esattamente come l'oggetto selezionato. Per aprire il pannello di allineamento è possibile anche cliccare su *Tools > Align > Align*.

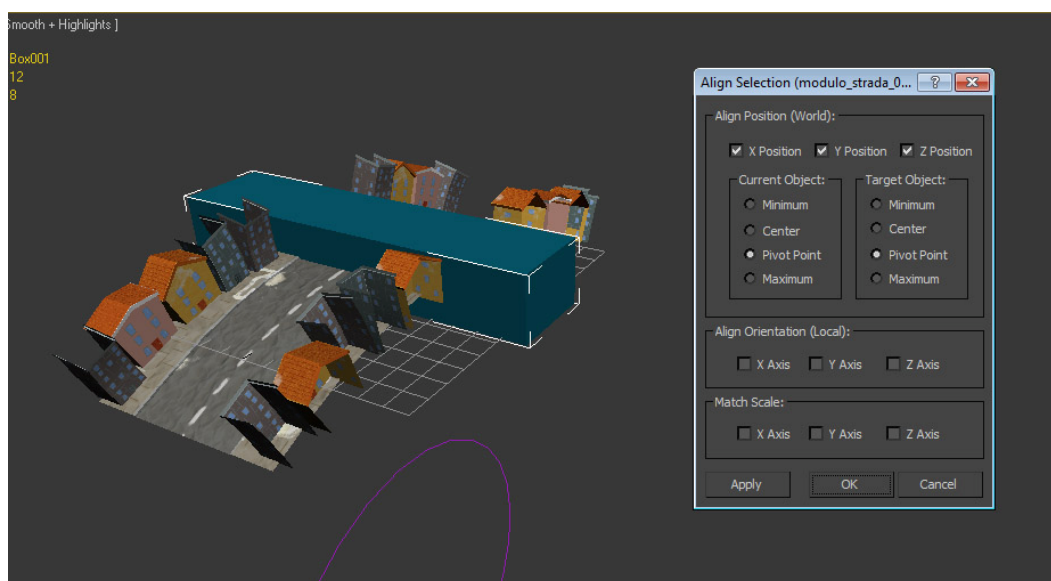


Fig. 4.19 Pannello Align

Nel caso in cui l'oggetto al quale vogliamo orientare la figura non sia più nella posizione originale esiste una funzione chiamata *Reset Xform* nel pannello delle *Utilities*. Questa funzione aggiungerà un modificatore al nostro oggetto *Editable Poly* che azzererà tutti i tipi di trasformazioni che ha l'oggetto, poichè ogni oggetto ha una storia di tutte le rotazioni e gli spostamenti effettuati che vengono memorizzati durante le trasformazioni che vengono eseguite.

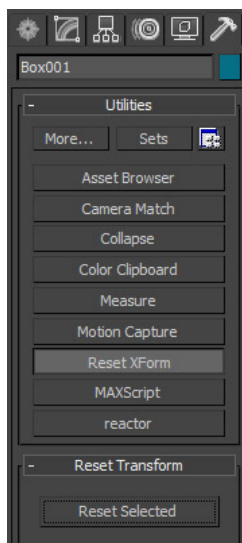


Fig. 4.20 Reset XForm

Una volta allineato la nostra figura al modulo aggiungere ulteriori segmenti alla nostra primitiva di base per poter creare successivamente gli archi del ponte. Aggiungendo ulteriori segmenti sulla lunghezza della figura si andranno a creare ulteriori poligoni sull'oggetto. Il numero dei segmenti da aggiungere dipende da quanti pilastri vorremo creare nel figura del ponte. In questo caso aggiungere quindi 4 segmenti per creare un pilastro al centro della strada e i due archi corrispondenti alle due corsie della strada.

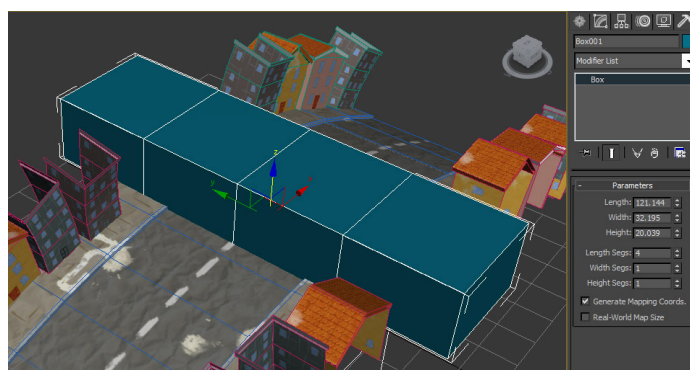


Fig. 4.21 Primitiva Box

Per creare l'estetica della figura simile a tutti gli oggetti del video gioco utilizzare il modificatore *Bend*, che è possibile aggiungere cliccando sul pannello *Modify* e selezionandolo poi dalla lista dei modificatori. La qualità di questo modificatore sarà determinata da quanti segmenti ho aggiunto alla figura di base. In seguito quindi aggiungere un ulteriore segmento per quanto riguarda la larghezza della

figura e poi tornando nel modificatore cliccare su asse X nel pannello *Bend Axis* e agendo sull'angolazione è possibile “piegarlo” per dare al ponte l'aspetto di tutto il gioco. Inoltre attraverso l'uso di una camera posta di fronte all'oggetto che si sta modellando si può notare come potrebbe essere la scena effettiva di gioco e in questo modo si potrà modellare meglio la figura di base.

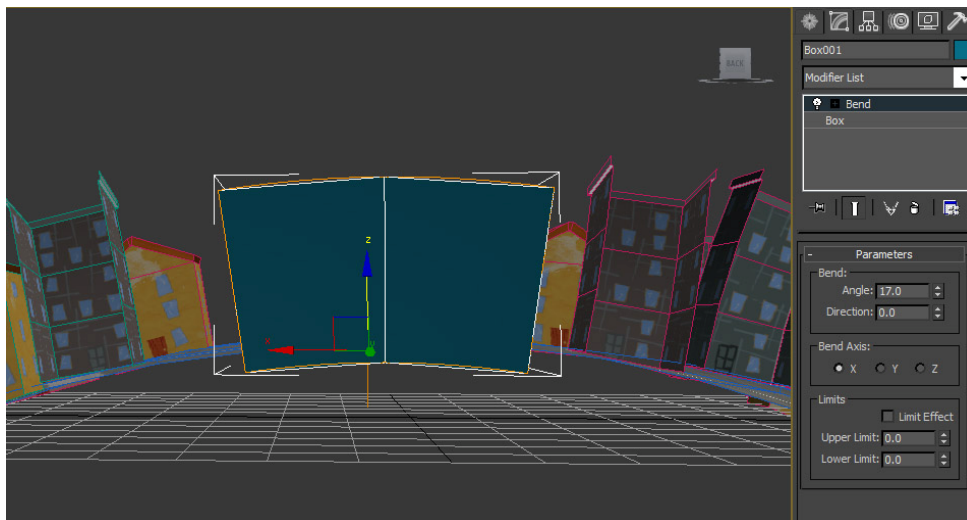


Fig. 4.22 Modificatore Bend

Quando si utilizza una camera attraverso la combinazione dei tasti SHIFT+F si attiverà il comando *Show Safe Frames* il quale mostrerà quello che in realtà si vede attraverso la *Viewport* della camera. Ciò è utile perchè durante la creazione del il gioco si dovrà decidere anche quale formato di risoluzione avrà. Se si apre la finestra delle impostazioni del rendering (F10) o *Render Setup* è possibile attivare il *Safe Frames* impostando la larghezza e l'altezza dell'*Output Size* nel pannello *Common Parameters*. Se ad esempio si vuole una risoluzione a 16:9 impostare come larghezza 1024 e come altezza 576. Inoltre questa funzione è utile perchè serve anche a capire le dimensioni dell'oggetto da modellare poichè al di fuori della *Viewport* della camera non si ha bisogno di modellare ulteriormente l'oggetto e ai fini della creazione di un video gioco questo accorgimento è importante perchè si dovranno utilizzare meno poligoni possibile per aumentare le prestazioni.

Se a questo punto le dimensioni della figura sono quelle desiderate aggiungere il modificatore *Edit Poly* attraverso il quale poi è possibile selezionare le facce o i poligoni dell' oggetto facendo attenzione a non modificare la geometria

dell'oggetto *box* al livello altrimenti potrebbe creare grossi problemi allo *stack* superiore della lista dei modificatori. Infatti utilizzando il modificatore *Edit Poly* è come se venisse creata una numerazione dei vertici, che qualora venga modificata nel livello più basso, potrebbe deformare la geometria della figura con risultati inattesi.

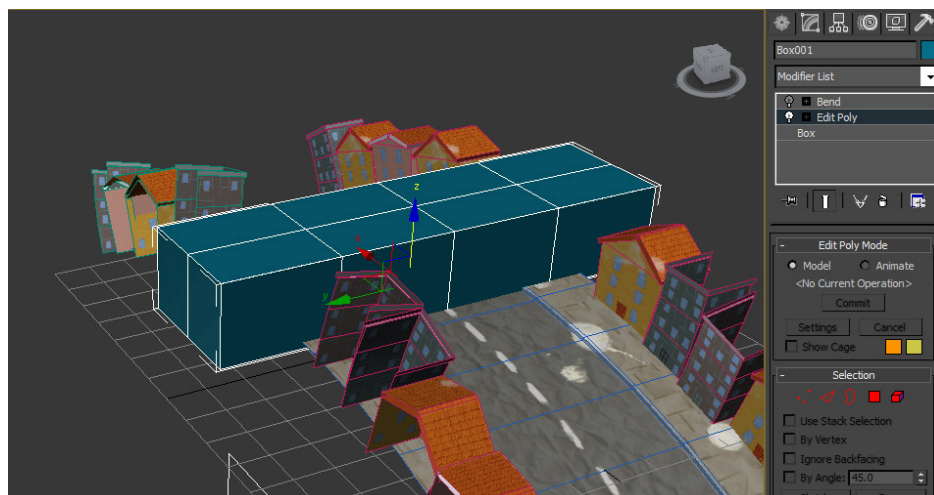


Fig. 4.23 Modificatore Edit Poly

A questo punto selezionando tutti i lati dei segmenti centrali relativi alla larghezza dell'oggetto utilizzare il comando *Chamfer* che è individuabile all'interno del pannello *Edit Edges*. Cliccando sul comando si aprirà una finestra di dialogo attraverso la quale regolare la quantità di smussatura poligonale dell'oggetto che in questo caso corrisponde al raddoppio dei segmenti.

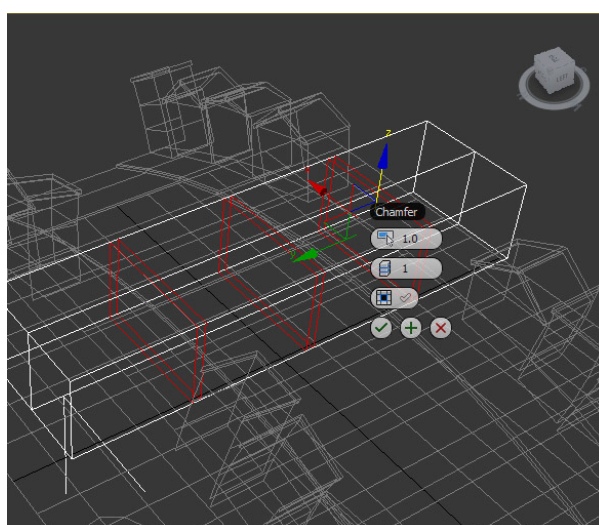


Fig. 4.24 Comando Chamfer

Per creare gli archi selezionera poi tutti i lati verticali corrispondenti alle facce della nostra figura e cliccando sul comando *Connect*, che è individuabile dentro al pannello *Edit Edges*, è possibile decidere quanti segmenti aggiungere sulla lunghezza dei lati selezionati e a quale altezza posizionarli sempre lungo l'asse dei lati. A differenza dei parametri di base attraverso i quali è possibile aggiungere nuovi segmenti, grazie al comando *Connect* si può decidere dove spostare i nuovi segmenti aggiunti alla figura. In questo aggiungere un solo segmento e posizionarlo in alto pari all'altezza che si vuole dare all'arco del ponte. Dopodichè creare le aperture relative agli archi selezionando le relative facce dei poligoni che poi è necessario cancellare. Ripetere l'operazione anche sulle facce alla base del ponte.

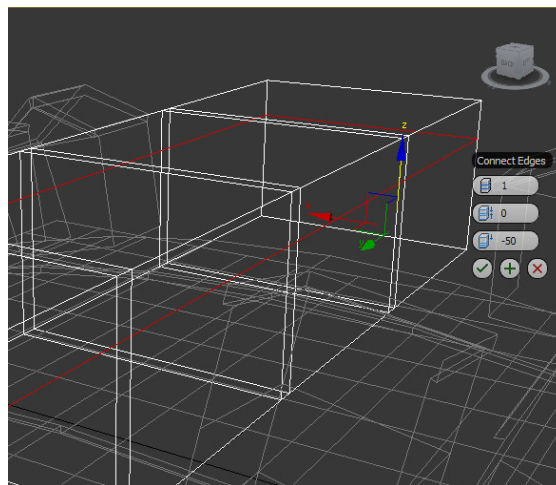


Fig. 4.25 Comando Connect

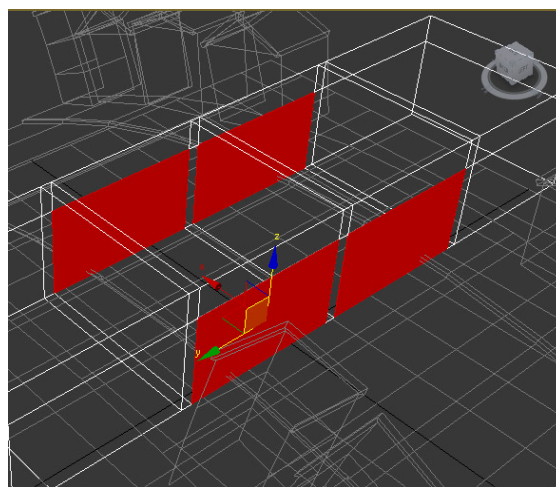


Fig. 4.26 Selezione poligoni

Andando ad eliminare alcune facce dei poligoni si creano degli spazi all'interno della figura che è necessario chiudere col comando *Bridge*, il quale crea una faccia poligonale fra due spigoli selezionati. Ripetere questa operazione più volte per creare i pilastri. Per la parte superiore usera invece i *Border* dal pannello *Edit Poly* e grazie al comando *Cap* chiudere le aperture sotto le travi del ponte.

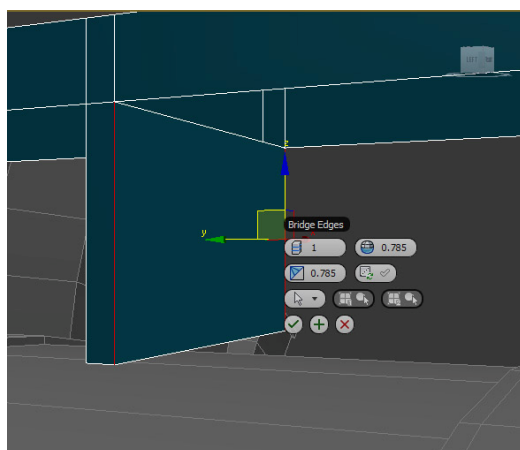


Fig. 4.27 Comando Bridge

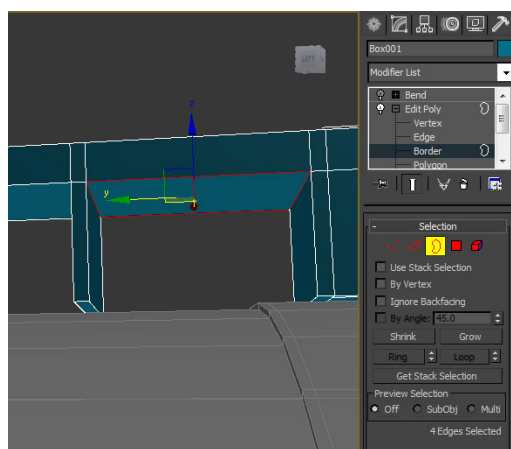


Fig. 4.28 Selezione Border e Comando Cap

Impostando la vista su *Left* selezionare gli spigoli relativi alla parte superiore degli archi. Poi utilizzare il comando *Connect* per aggiungere un segmento al centro e attraverso il modificatore *Chamfer* raddoppiare il segmento creato allargando la distanza della quantità voluta. Si ottiene così la struttura principale del ponte che da rifinire. Selezionando i vertici relativi al punto di congiunzione con gli archi e abbassandoli è possibile creare gli archi stilizzati. Rilezionare gli spigoli superiori e utilizzando il comando *Connect* inserire un segmento.

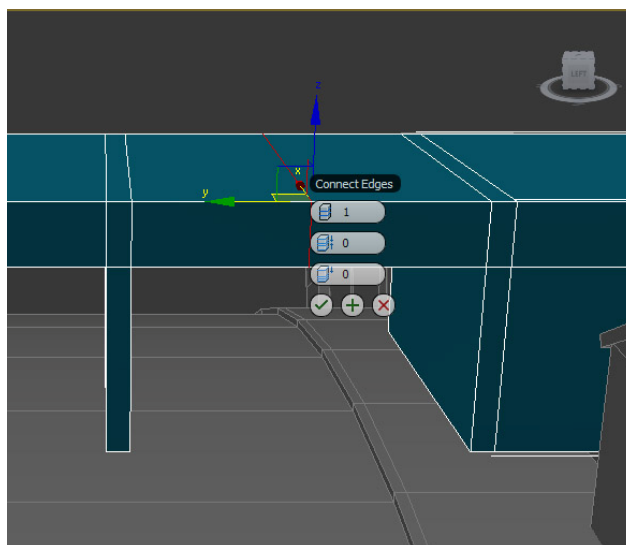


Fig. 4.29 Comando Connect Edges

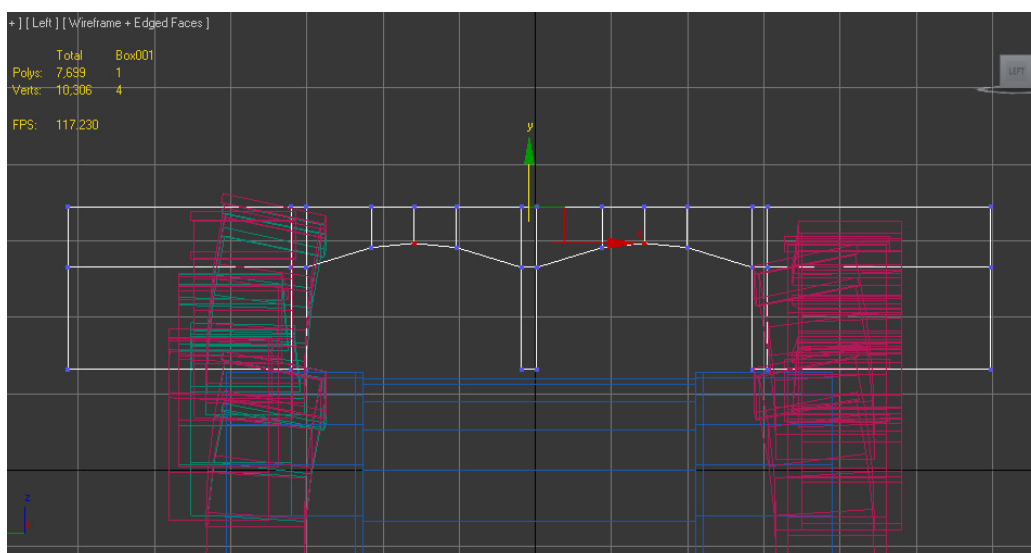


Fig. 4.30 Creazione degli archi

Ora selezionare tutti i segmenti orizzontali della figura per applicare un nuovi segmenti attraverso il comando *Connect* ed eseguire poi il *Bend* del poligono facendo attenzione a non avere selezionato nessuna sottocategoria nel modificatore *Edit Poly* (ad esempio potrebbero essere selezionati i vertici della *Editable Poly*). Aggiungere quattro segmenti col comando *Connect* per piegare l'oggetto attraverso il comando *Bend* con una angolazione di circa 15 gradi sull'asse x del *Bend*. Applicare un'altro modificatore *Bend* sull'asse y con una direzione di 90 gradi e un angolo di circa -28 gradi in modo da piegare il ponte con le estremità laterali verso il basso. La parte relativa alla struttura principale

del ponte è conclusa ma si vuole modellare anche una ringhiera lungo i bordi che simuli un parapetto.

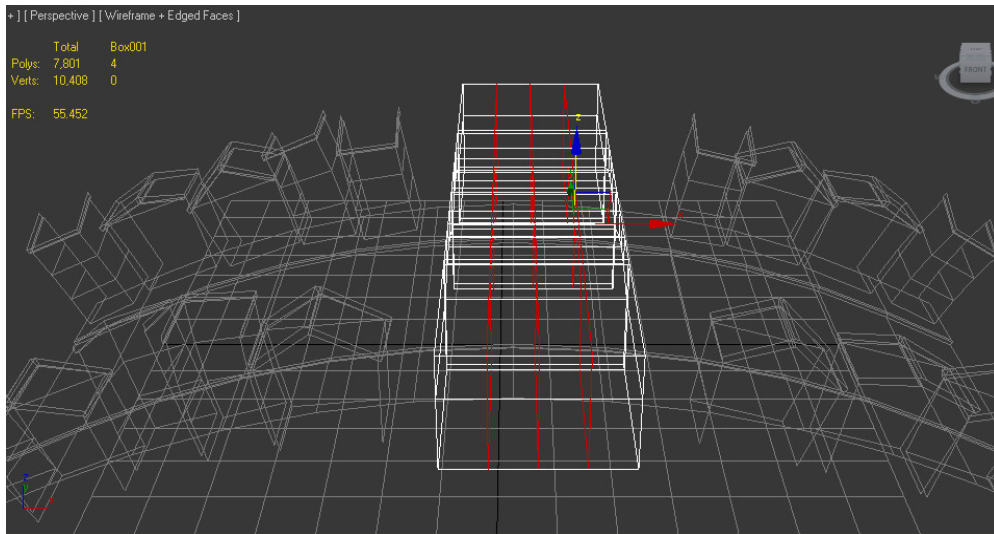


Fig. 4.31 Inserimento dei segmenti

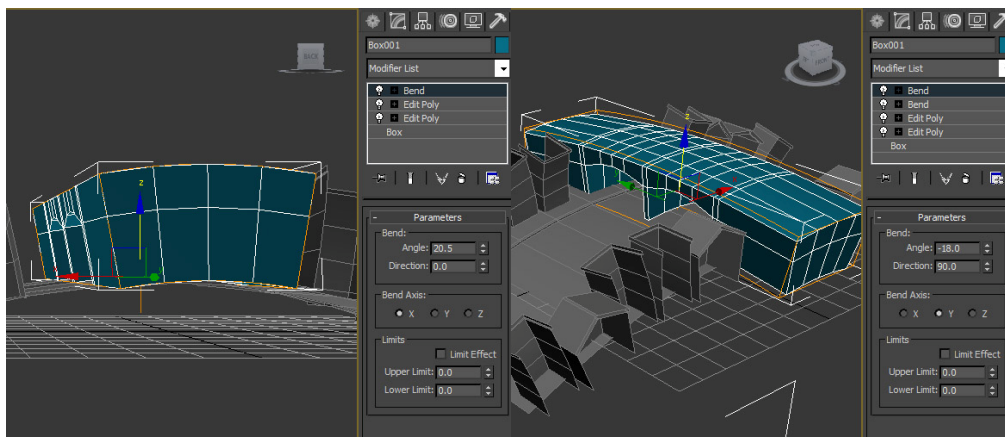


Fig. 4.32 Modificatore Bend

Ora per poter lavorare sul modello creato e poter aggiungere dettagli le strade sono due: la prima è convertire l'oggetto in *Editable Poly* cliccando col tasto destro e selezionando l'opzione relativa dal menu a discesa, la seconda invece è quella di aggiungere un ulteriore modificatore *Edit Poly* in cima alla lista in modo che si possa lavorare sulla forma creata mantenendo gli effetti creati dal *bending*, ma avendo comunque la possibilità di tornare indietro e modellare ancora la struttura. In questo modo nell'*Edit Poly* che è stato aggiunto non si modificherà niente ma serve solo per poter selezionare gli spigoli già piegati aventi la forma della struttura del ponte.

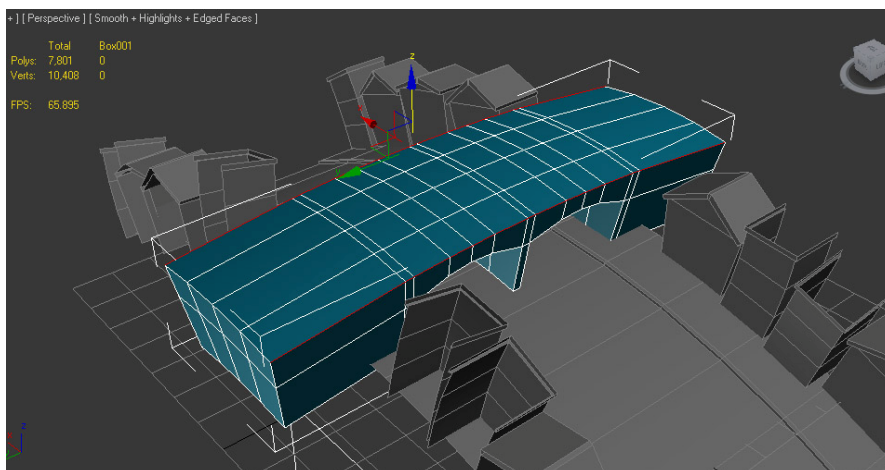


Fig. 4.33 Selezione degli spigoli laterali

Una volta selezionati gli spigoli sia da un lato che dall'altro del ponte usare il comando *Create Shape* che si trova nel pannello *Edit Edges*. Dopo aver cliccato si aprirà un pannello con due opzioni riguardo allo *Shape Type* di cui selezionare *Linear* per essere sicuri che la forma selezionata sia esattamente uguale a quella selezionata. Assegnare il nome "Railings" e cliccare OK. In questo modo il programma andrà a creare due *Spline* della forma relativa agli spigoli che sono stati selezionati.

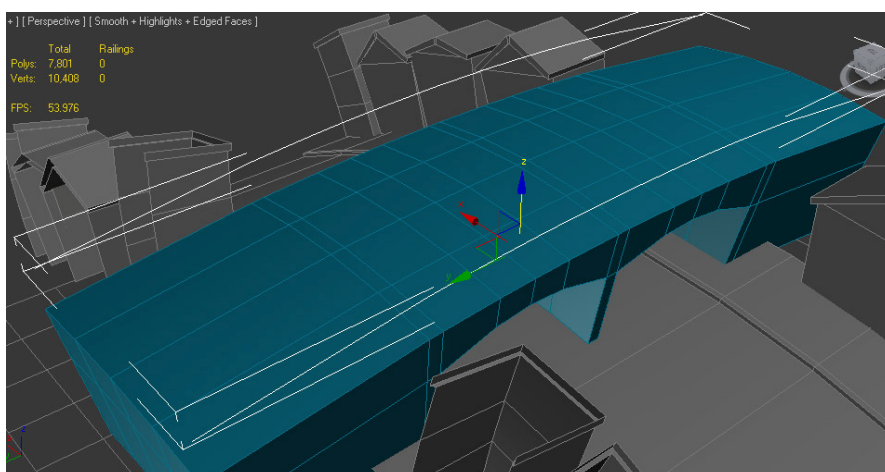


Fig. 4.34 Creazione delle spline

Sulle spline create bisognerà applicare delle trasformazioni per modificare la scalatura ma allo stesso tempo mantenere la giusta curvatura rispetto al ponte in modo che la ringhiera del parapetto rispecchi l'andamento degli spigoli della struttura principale. Quindi aggiungere il modificatore *Sweep* grazie al quale si potranno assegnare alla spline una serie di profili per poter ottenere la forma

voluta. In questo caso se si vuole ottenere la forma di un tubo selezionare *Cylinder* dal menù a discesa di *Built-In Section* del pannello *Section Type* e poi impostare il valore di *Radius* in *Parameters* a circa 0.5 e il valore di *Steps* in *Interpolation* ad 1. L'aspetto più significativo di questo modificatore è il pannello *Sweep Parameters* che permetterà di spostare l'oggetto nello spazio e di creare un offset della spline rispetto al punto di partenza variando la scalatura dell'oggetto ma soprattutto mantenendo la curvatura originale. Agendo sul valore della *y* spostare la *spline* ad un'altezza maggiore rispetto allo spigolo della struttura del ponte.

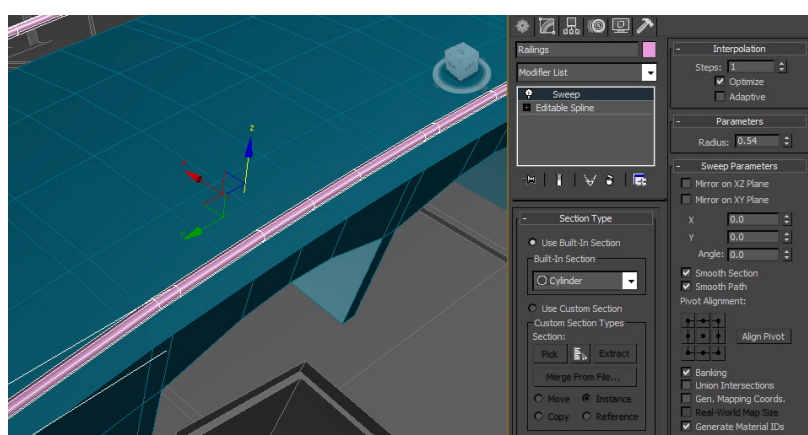


Fig. 4.35 Modificatore Sweep

Infine si vorranno creare i “paletti” del parapetto che faranno da supporto alle ringhiere che sono state appena create. Per creare il primo paletto modellare un singolo box di dimensioni 1 di lunghezza, 1 di larghezza e 10 di altezza. Poi dal pannello *Extras*, tra le varie funzioni *Array* a disposizione, aprire *Spacing Tool* il quale permette di distribuire gli oggetti su una spline. Poiché il tubo che è stato creato non è un path ma viene vista come geometria allora procedere in questo modo: prima copiare la *spline* in un altro oggetto chiamandolo “Riferimento Paletti” e cancellandogli il modificatore *Sweep*, poi aprire *Spacing Tool*, selezionare il paletto e cliccando su *Pick Path* dal pannello poi selezionare la *spline*. In questo modo verrà posizionato il *box* relativo al paletto esattamente sulla *spline*. Attraverso il parametro *Count* è possibile aggiungerne altri a piacere. Dopodiché si dovrà impostare la geometria in modo che segua la normale della *spline* poiché il posizionamento dei paletti deve seguire la curvatura della struttura del ponte. Attivare quindi il checkbox *Follow* nella sezione *Context* del pannello

Spacing Tool. Infine impostare con *Type of Object* al valore *Instance*. Questo farà sì che le trasformazioni ad uno dei *box* relativi ai paletti saranno ereditate da tutti gli altri.

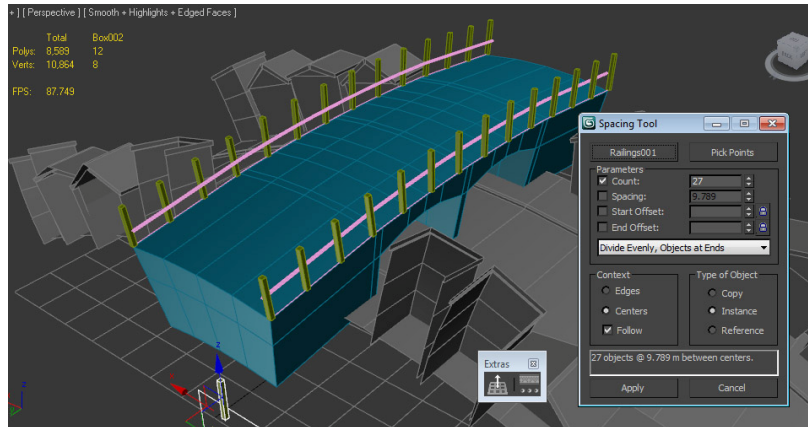


Fig. 4.36 Spacing Tool

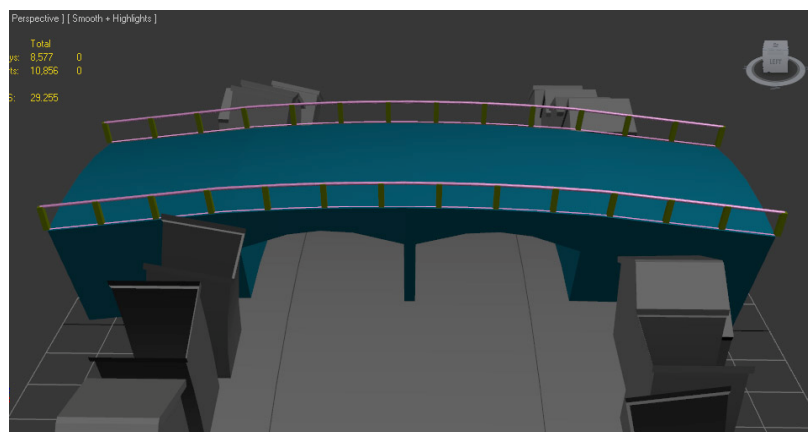


Fig. 4.37 Modello finale del ponte

4.2.3 Modellazione di un Animale

Per creare il modello di uno degli animali presenti nella scena di gioco della campagna si è proceduto nel modo seguente. Per il corpo dell'animale si parte dalla primitiva estesa *Oil-Tank* che andrà a creare un oggetto con la tipica forma di una capsula. Una volta creata la forma impostare i seguenti parametri alla primitiva: *Sides* a 12 e *Height Segments* a 1 e in seguito aggiungere un modificatore *Edit Poly*. Aprendo il modificatore *Edit Poly* si dovranno selezionare i vertici e cliccare su quelli presenti in figura. Selezionando il primo vertice in cima alla capsula e usando il comando *Soft Selection* vengono modificati anche i

vertici nelle vicinanze con un valore che decrescerà in base alla lontananza dei vertici. Nella *Soft Selection* si dovranno aggiungere i seguenti parametri: spuntare *Affect Back Facing* con un *Fall Off* di circa 12. Lo stesso metodo sarà da impostare nel vertice opposto a quello selezionato spostando e scalando i vertici adiacenti come in figura. In seguito si dovrà aggiungere il modificatore *Smooth* per ammorbidire il nostro oggetto e smussare tutte le sue forme.

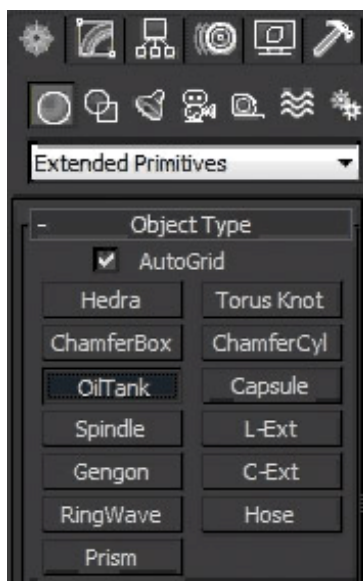


Fig. 4.38 Extended Primitives

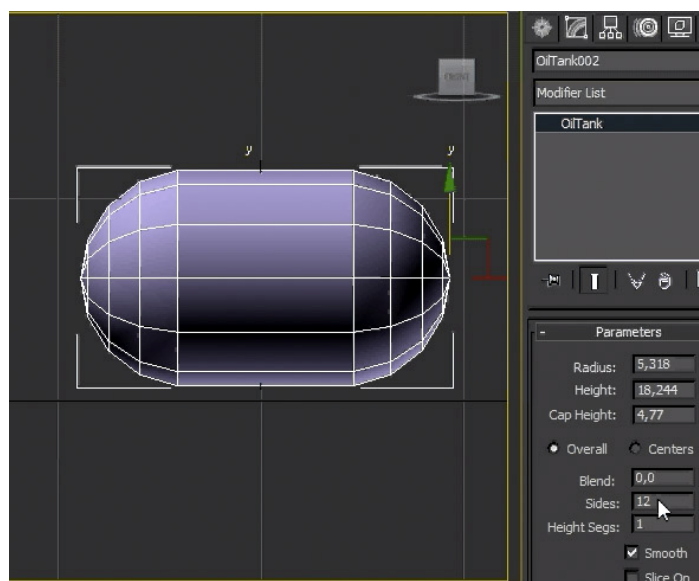


Fig. 4.39 OilTank

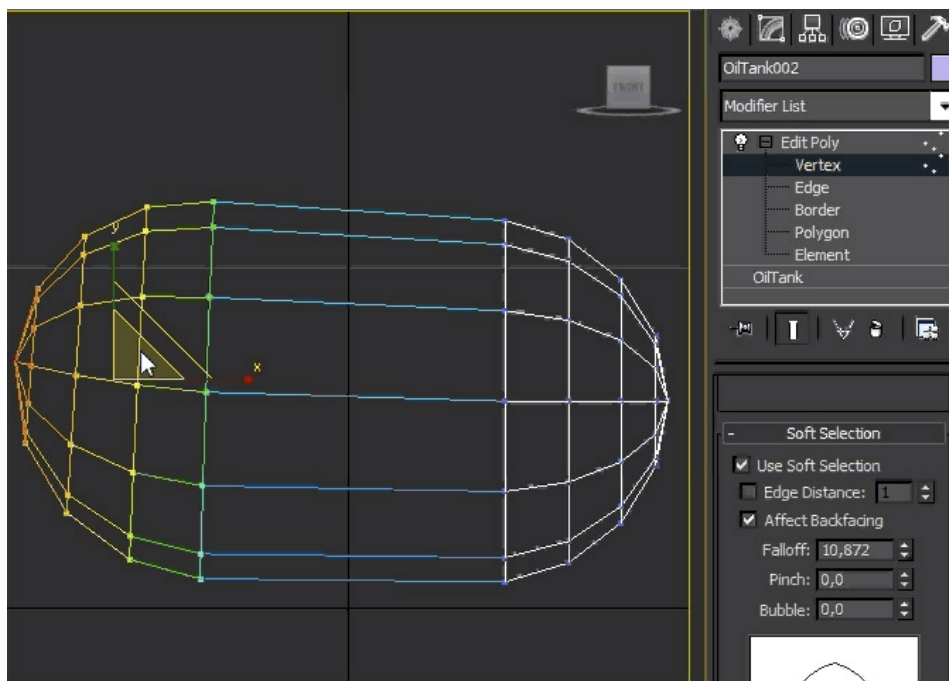


Fig. 4.40 Soft Selection

Il passo successivo sarà quello di creare la testa dell'animale seguendo le seguenti fasi. Prima clonare l'oggetto appena creato e spostare la testa creata posizionandola vicino al collo dell'animale ruotandola e scalandola.

In seguito aggiungere il modificatore *Taper* il quale permette di deformare (allargare o stringere) la forma in questione all'interno di una primitiva corrispondente ad una piramide. Infatti è necessario stringere la testa dell'animale verso la fronte. Per poter fare ciò bisognerà usare il *Gizmo* interno al modificatore. Poi attraverso un'altro comando del modificatore avremo la possibilità di arrotondare i poligoni verso l'interno in modo da ottenere una curvatura. I parametri da assegnare per il pannello *Taper* sono di circa 0,58 per *Amount* e di -0,42 per *Curve* e poi selezionare l'asse *Z* come *Primary* ed *XY* per quanto riguarda *Effect*.

All'interno di *Edit Poly* selezionare *Edge* e attraverso il comando *Connect* unire gli spigoli centrali della capsula. Una volta fatto ciò scalare la testa allargandola della quantità desiderata come da figura sempre attraverso il modificatore *Taper*.

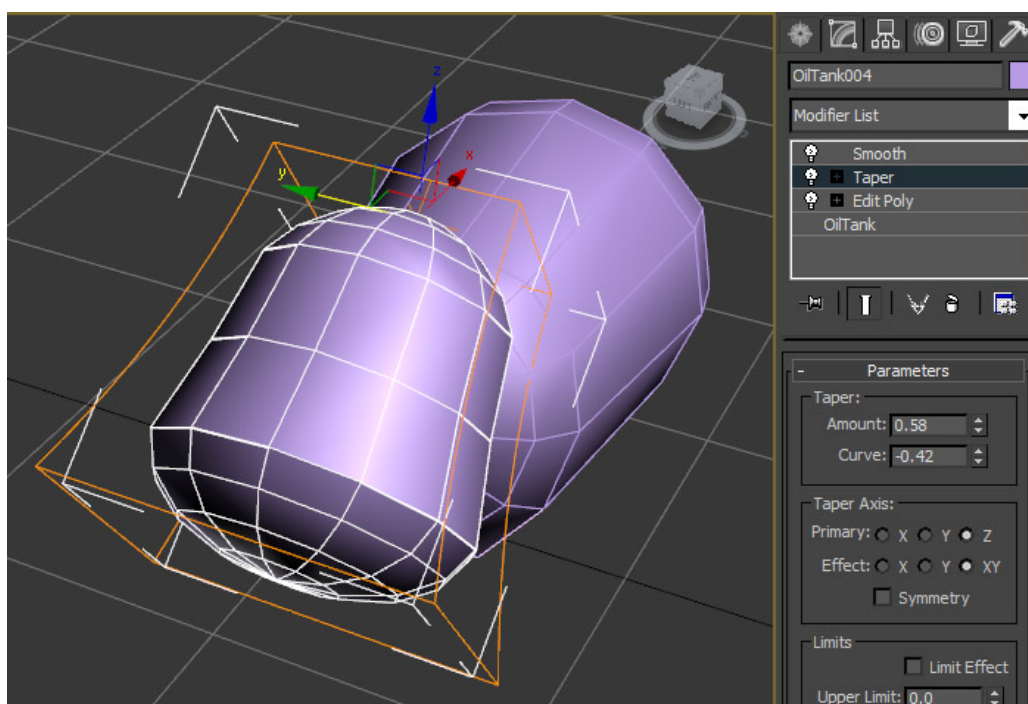


Fig. 4.41 Modificatore Taper e Parametri

Per quanto riguarda le corna della testa dell'animale si fa uso di una primitiva semplice il Cono. Questa forma ha tre parametri principali: due raggi e l'altezza. I

parametri da assegnare sono i seguenti: *Radius1* pari a 1, *Radius2* pari a 0.2, ed altezza pari a 7 circa. In seguito si dovrà aggiungere il modificatore *Bend*. Nei parametri assegnare un angolo di 82 e una direzione pari a 90 sull'asse zeta.

In seguito posizionare il corno creato sulla testa dell'animale ruotandolo ed allineandolo nella giusta posizione e diminuire i numeri di poligoni utilizzati riducendo il parametro *Sides* da 12 a 24.

Infine utilizzare il modificatore *Symmetry* per avere una copia esatta del secondo corno. Dentro il modificatore *Symmetry* selezionare *Mirror* e nei suoi parametri interni l'asse Y.

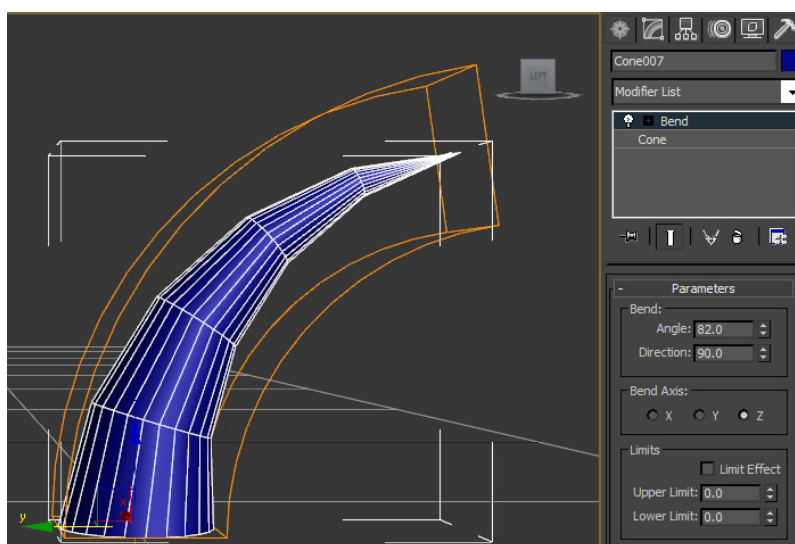


Fig. 4.42 Primitiva Cone con modificatore Bend

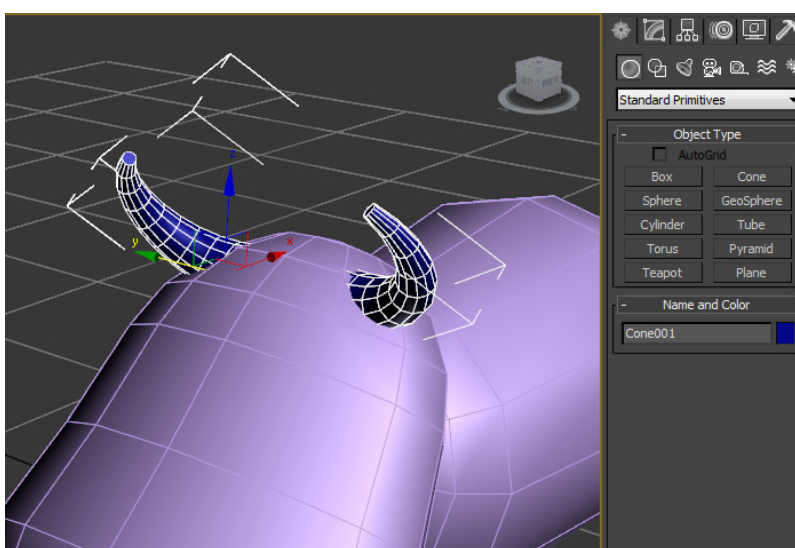


Fig. 4.43 Corna dell'animale posizionate sulla testa

La creazione delle orecchie dell'animale parte invece dalla primitiva *Box*. Una volta creato un piccolo *Box* posizionarlo sulla testa in prossimità delle corna. Aggiungere il modificatore *Edit Poly* al *Box* e selezionare *Vertices*. Spostare i vertici superiori del *Box* in modo da ottenere una forma somigliante ad un orecchio. Questo procedimento si ottiene selezionando gli spigoli superiori ed inferiori dell'orecchio e attraverso il comando *Connect* aggiungendo nuovi vertici che si dovranno spostare per perfezionare la forma desiderata. Selezionare tutti i poligoni della faccia anteriore dell'orecchio e attraverso il comando *Bevel*, impostando un valore di $-0,372$, si ottiene una depressione sulla superficie. Attivare ora il comando *Extrude*, selezionare *Local Normals* ed assegnando un valore di circa $0,362$ su tutte le facce laterali dell'orecchio si otterrà una estrusione dei bordi (contorno dell'orecchio).

Scalare poi l'orecchio della dimensione necessaria e posizionarlo vicino alle corna. Selezionando la faccia centrale dei poligoni posteriori dell'orecchio attraverso la *Soft Selection* si spostare di poco la superficie sull'asse X. Infine applicare il modificatore *Smooth* e il modificatore *Symmetry* per creare il secondo orecchio.

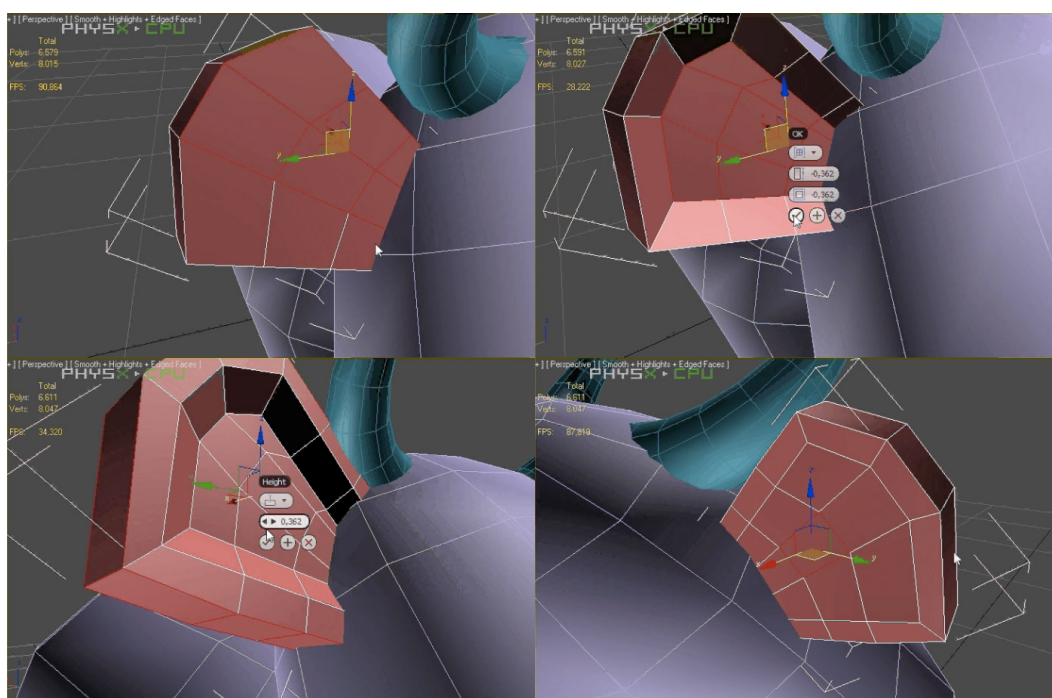


Fig. 4.44 Processo di creazione delle orecchie dell'animale

Si dovranno ora creare i materiali da assegnare alla figura. Per il corpo e la testa dell'animale usare il materiale *Diffuse* di colore bianco mentre per le orecchie selezionare i poligoni interni all'orecchio (come da figura) ed assegnare un nuovo materiale *Diffuse* di colore rosa, mentre per la parte esterna dell'orecchio selezionare il materiale di colore bianco. Questo procedimento si ottiene attraverso la creazione di un multimateriale a cui saranno assegnati due ID: il bianco avrà ID 1 mentre il rosa ID 2. Per assegnare l'ID giusto ai poligoni che si vorranno colorare bisogna procedere aggiungendo un modificatore *Edit Mesh* e nei parametri del pannello *Surface Properties* assegnare Id 2 al materiale delle *mesh* selezionate. Con lo stesso procedimento si colora la testa dell'animale.

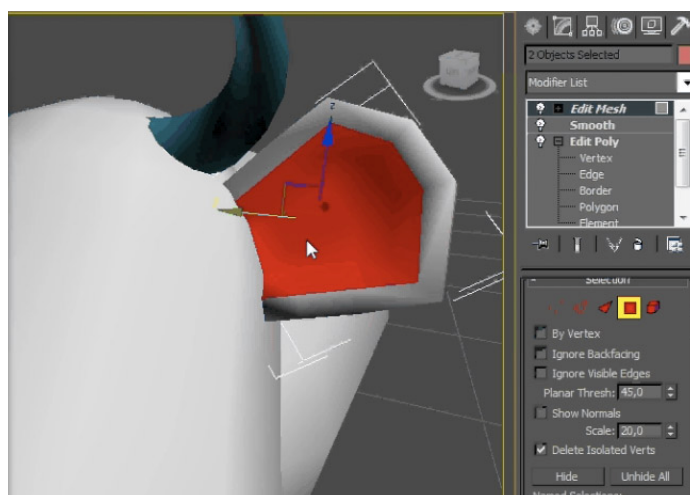


Fig. 4.45 Selezione dei poligoni interni all'orecchio

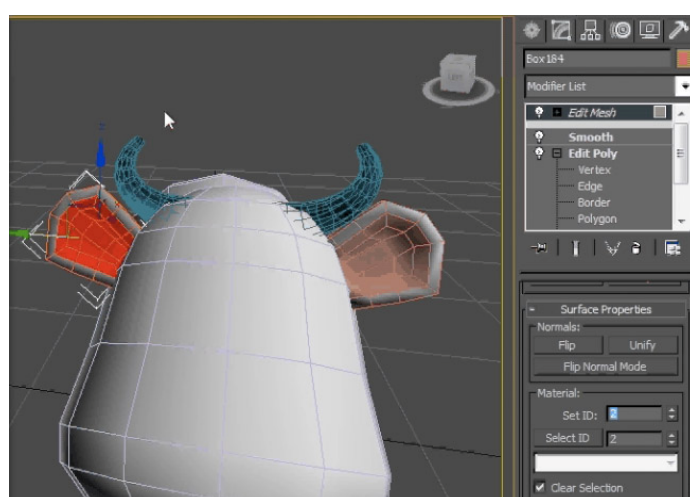
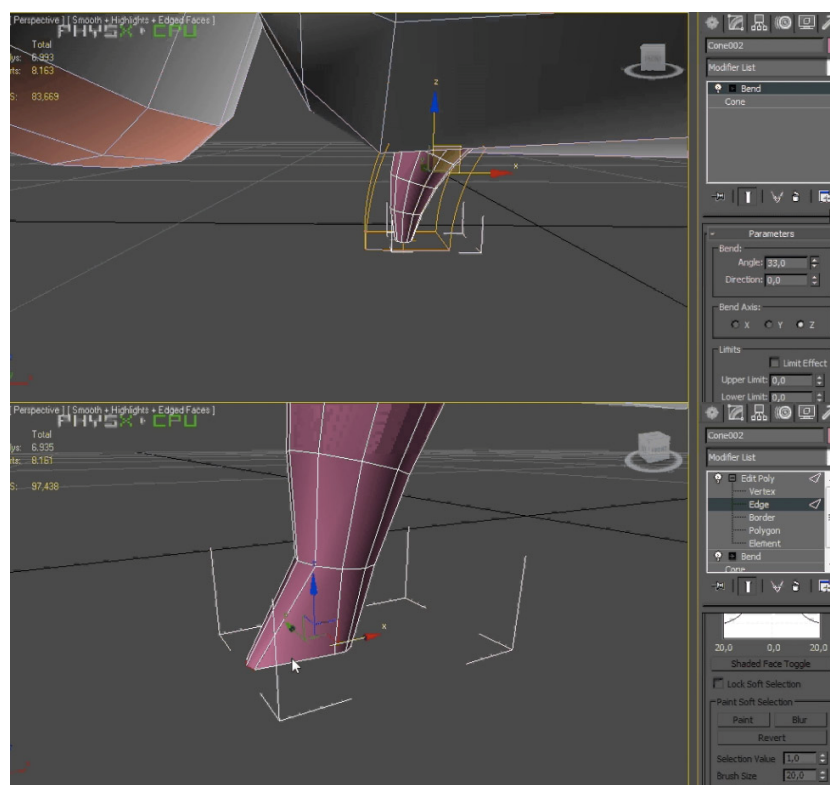


Fig. 4.46 Assegnazione al materiale con ID 2 ai poligoni interni

Per realizzare le gambe si dovrà partire dalla primitiva Cono, assegnando a *Radius1* un valore di circa 0,2 mentre a *Radius2* un valore di 0,9, un'altezza di circa 3, 4 segmenti e 10 *sides*. Posizionare la gamba creata sotto il corpo dell'animale e in seguito assegnare il modificatore *Bend* con un valore dell'angolo di circa 35. Assegnare ora il modificatore *Edit Poly* alla gamba per modellare la zampa. Selezionare *Edge* dal modificatore e cliccare lo spigolo anteriore della parte più stretta del cono, per dare la prima forma alla futura zampa. Infine selezionare i poligoni sottostanti alla zampa e allargare la scalatura per plasmare e migliorare la forma della zampa. Successivamente, assegnando a *Taper* il valore di circa 0,21 per *Amount* e di 2,65 per *Curve*, ingrandire secondo le necessità tutta la gamba. Copiare e clonare l'intero oggetto per ottenere le altre tre gambe. Assegneremo materiale di colore bianco alla gambe e di colore nero per i piedi attraverso la funzione *Multimaterial*.



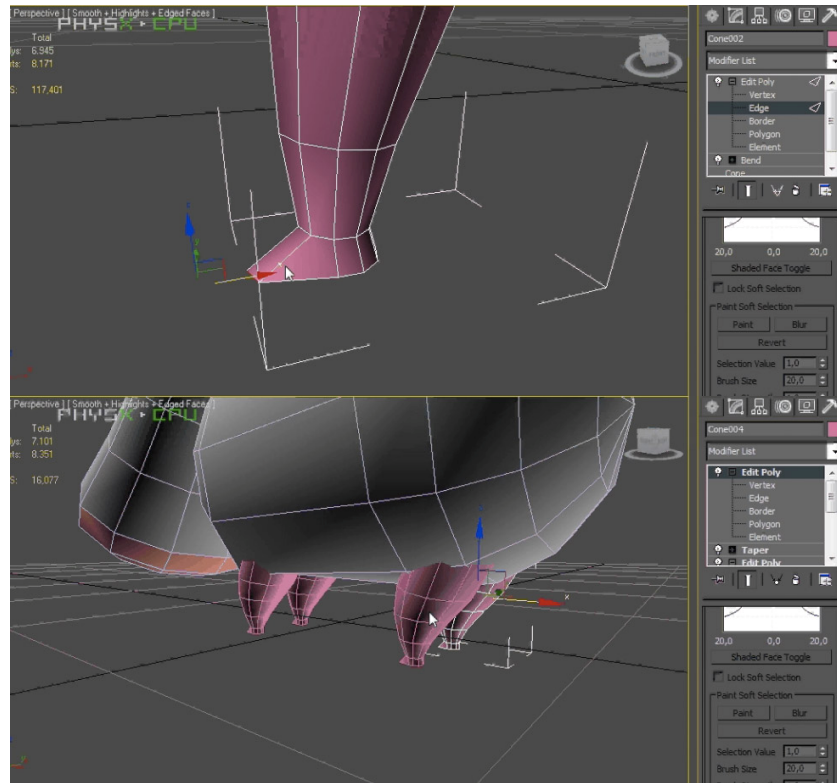


Fig. 4.47 Processo di creazione della gambe

Per modellare la coda si parte dalla primitiva Cilindro con un raggio di 0,18 e una altezza di circa 5, mentre il parametro *Sides* sarà da impostare a 6.

Aggiungere il modificatore *Edit Poly* e eliminare il poligono superiore del cilindro. Applicare il comando *Extrude* due volte di seguito per collegare la coda al corpo dell'animale. Selezionando poi i vertici, impostare la curvatura voluta al cilindro della coda e tramite il comando *Soft Selection* allargare l'inizio della coda, stringere la fine. Applicare poi il modificatore *Smooth*.

Infine selezionando *Border* dal pannello *Edit Poly* estruderemo quattro file di poligoni per ottenere i ciuffi della coda e applicando *Cap* chiudere il punto finale. Assegnare il materiale bianco alla coda e materiale nero al ciuffo.

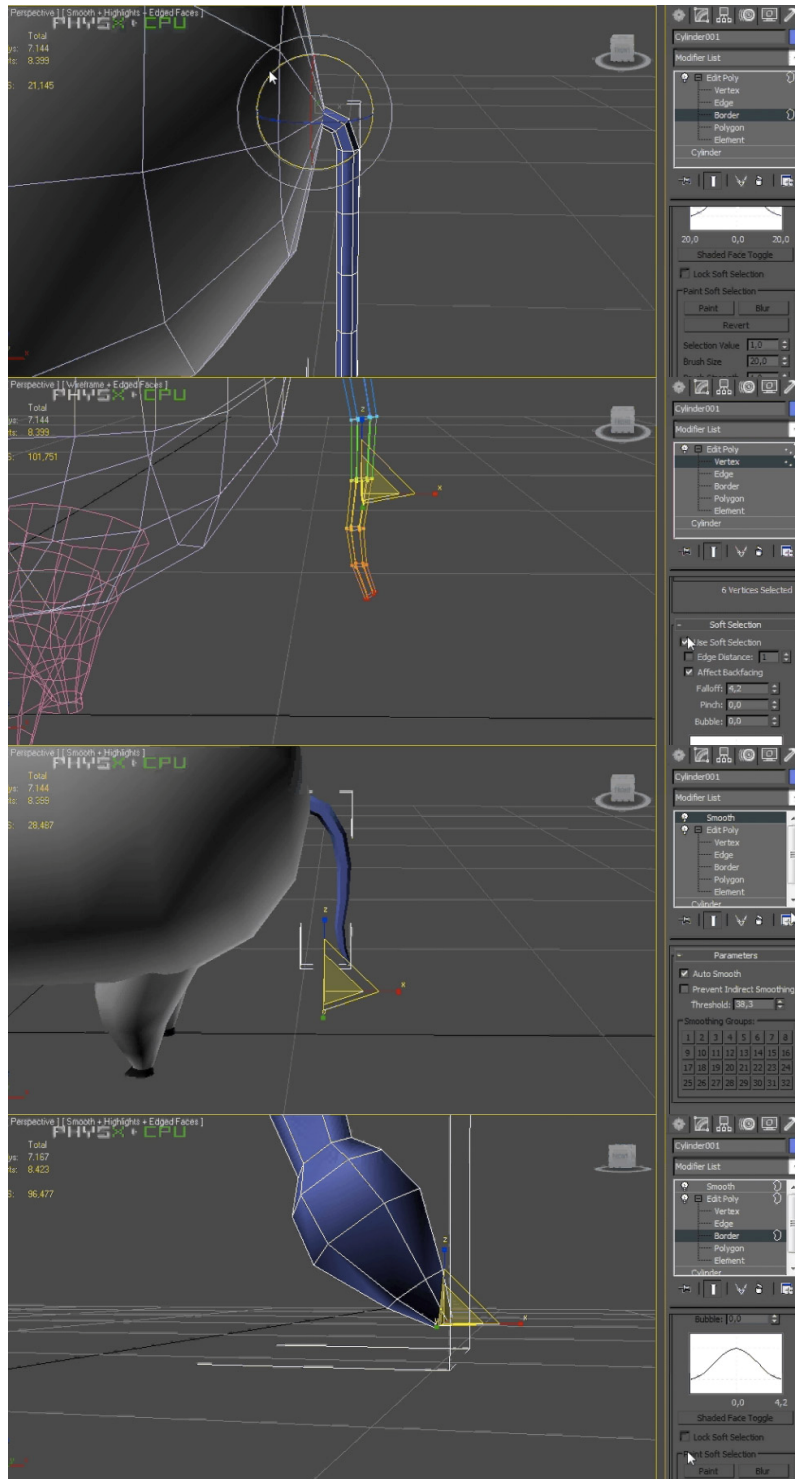


Fig. 4.49 Processo di creazione della coda dell'animale

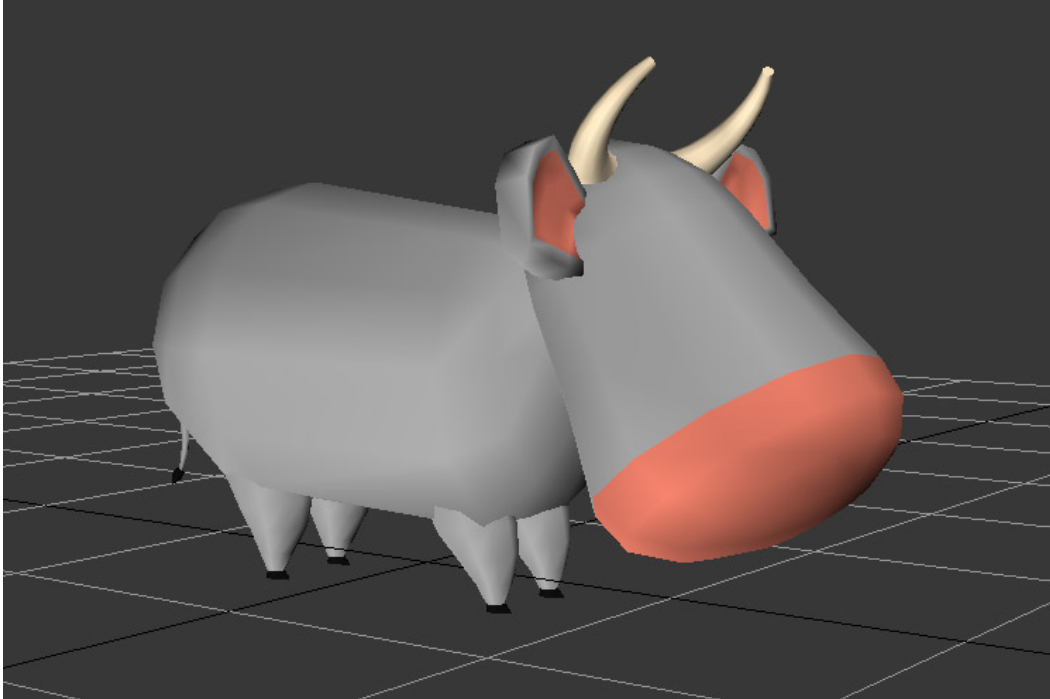


Fig. 4.50 Modello finale dell'animale

4.2.4 Modellazione, Pelle, Ossatura ed Animazione del Personaggio

Prima di poter procedere all'animazione di un personaggio 3D è necessario creare una *mesh* 3D statica. In seguito deve essere aggiunto un sistema di articolazione e ossature in modo che l'animatore possa mettere in posa il modello quindi il processo successivo alla modellazione è quello di prepararlo all'animazione attraverso un processo chiamato *Rigging*. Questa tecnica corrisponde essenzialmente ad un'ossatura collegata alla mesh 3D. Questa struttura è costituita da ossa e articolazioni, ognuna delle quali agisce come un oggetto che gli animatori possono utilizzare per manipolare il personaggio nella posa voluta. Una ossatura è una gerarchia capace di deformare la mesh alla base di un oggetto tridimensionale. Le ossature sono in grado di modificare le mesh deformandole in base al loro movimento. Una mesh deformata prende il nome di *Skin* (pelle). Normalmente un'ossatura viene creata dopo la mesh della pelle, utilizzando questa ultima come riferimento. Infatti per creare l'animazione del personaggio interno al gioco è stata creata prima l'intera modellazione della mesh.

Modellazione

Per la modellazione del corpo del personaggio sono state utilizzate due primitive semplici di tipo *Sphere*, dalle quali sono state ricavate due emisfere una di diametro leggermente più grande dell'altra, che in seguito sono state congiunte. Per collegare le due semisfere selezionare i bordi attraverso il comando *Bridge* per congiungere le due semisfere. Successivamente attraverso il comando scalatura modificare leggermente la forma dell'oggetto stringendolo verso il centro e deformandolo verso l'alto.

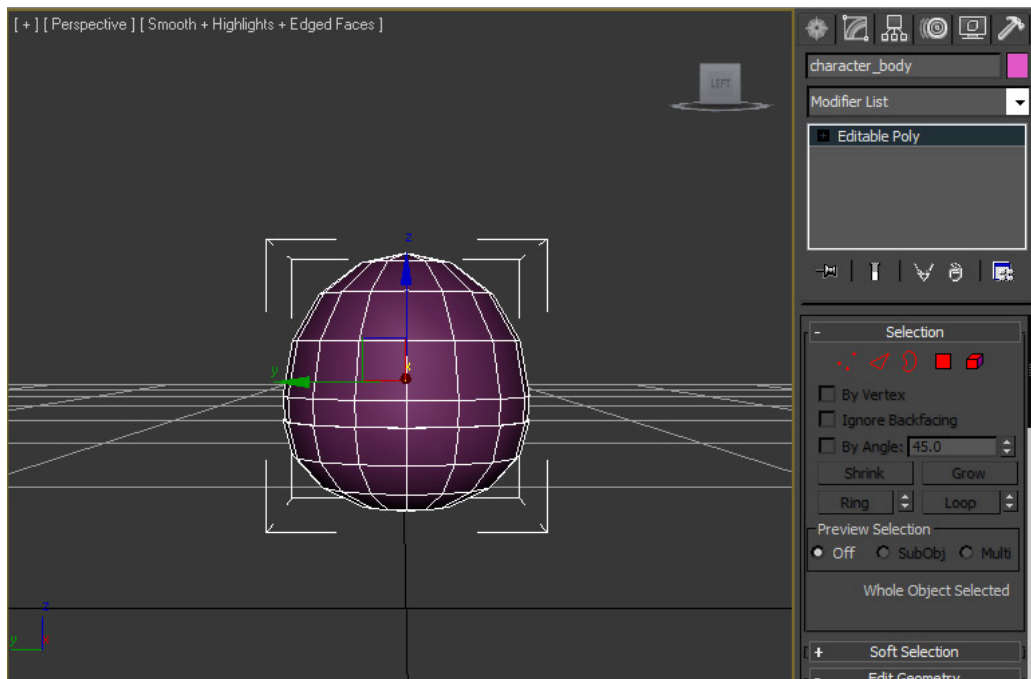


Fig. 4.51 Creazione del corpo

Per le gambe e le braccia sono state utilizzate delle primitive di tipo Cono mentre per quanto riguarda i piedi invece si inizia da una primitiva di tipo Box, con 3 segmenti in lunghezza, 2 segmenti in larghezza, e 1 segmento in altezza.

In seguito aggiungere a questa primitiva un modificatore di tipo *Edit Poly*. Selezionare poi i vertici nell'*Edit Poly* e spostarli per ottenere la forma come da figura, in seguito aggiungere il modificatore *Turbo Smooth*.

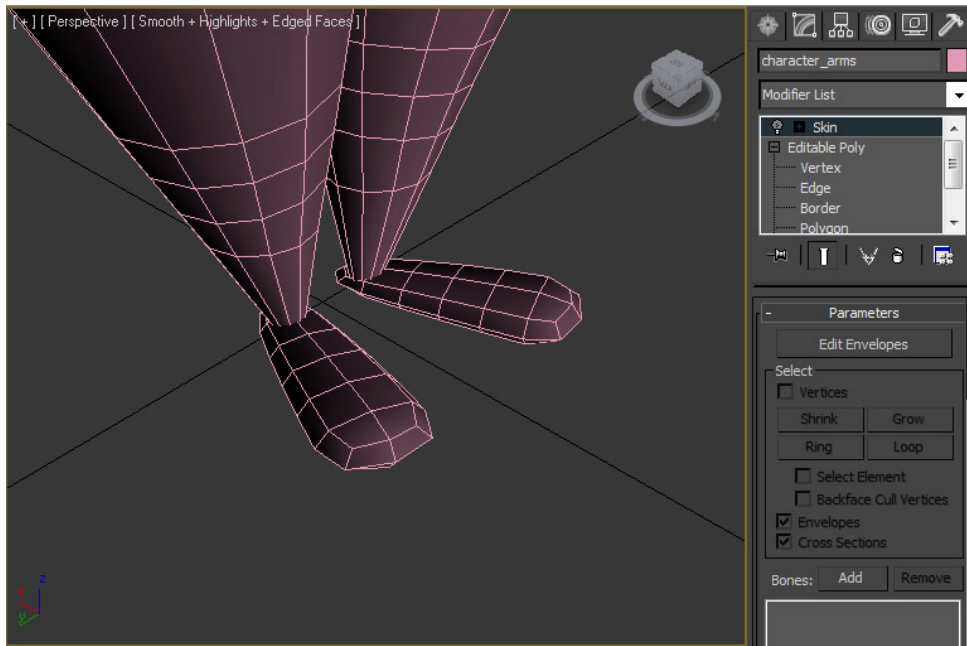


Fig.4.52 Creazione dei piedi

Ora si creerà la testa. Per la sua modellazione si parte da una primitiva di tipo *Box* con due segmenti in lunghezza, due in larghezza, due in altezza e per quanto riguarda i parametri larghezza, lunghezza e altezza impostare il valore a 4. Successivamente si applicheranno i modificatori *Edit Poly* e poi *Spherify* con una percentuale pari a 65.

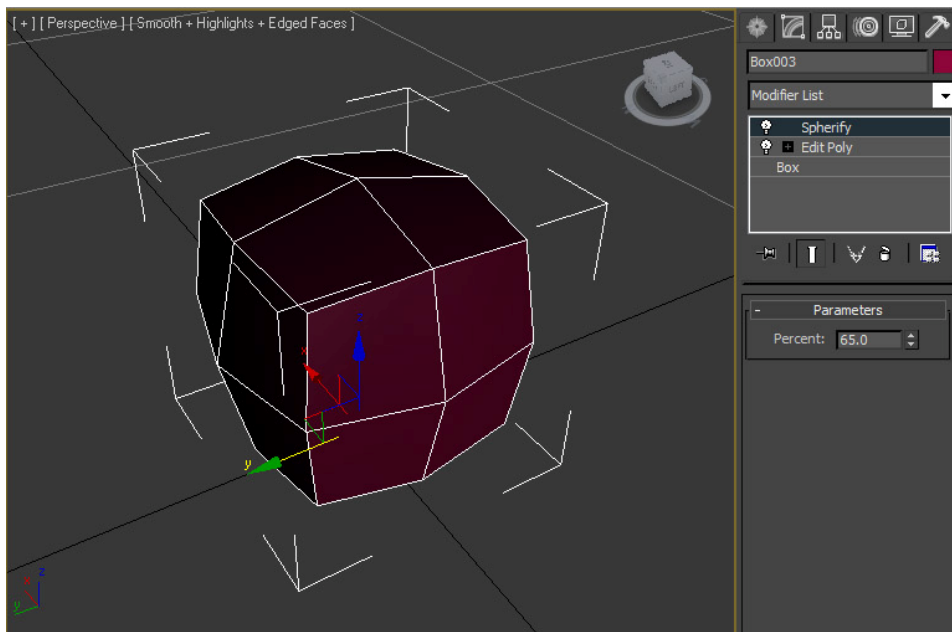


Fig. 4.53 Creazione della testa

Applicare il modificatore *Turbo Smooth* per aumentare i poligoni della testa. In seguito andrà creato un altro modificatore Edit Poly per poter selezionare le superfici con le quali creare i capelli. A questo punto utilizzando la visuale Left, selezionare i poligoni con i quali si modelleranno i capelli. Clonare l'elemento selezionato per poter avere a disposizione un nuovo oggetto. Applicare il modificatore *Shell* con i seguenti parametri: numero di segmenti pari a 1, *Inner Amount* pari a 0 ed *Outer Amount* pari a 0.3. Una volta terminata la modellazione si procederà alla sua ossatura.

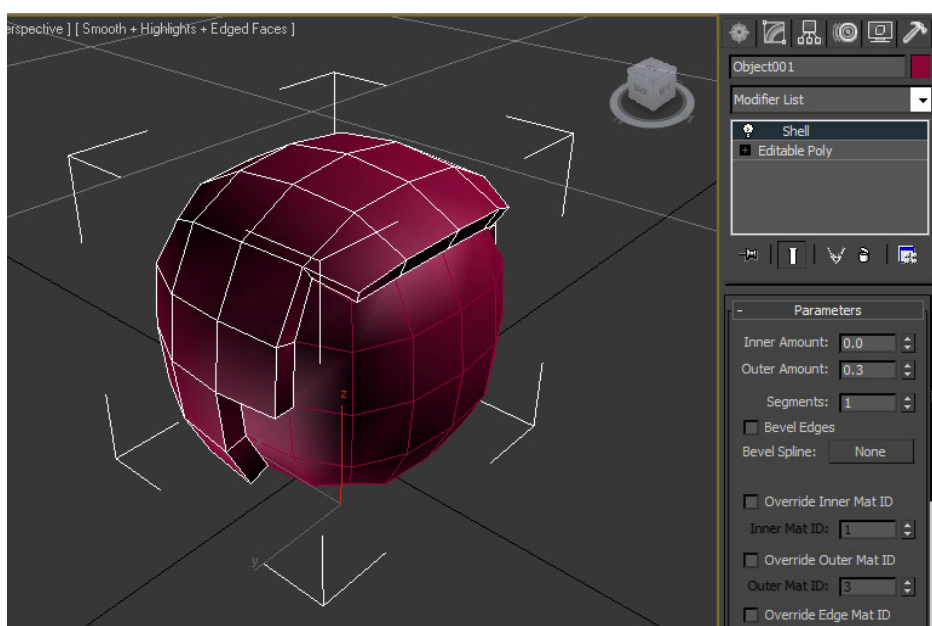


Fig. 4.54 Creazione dei capelli

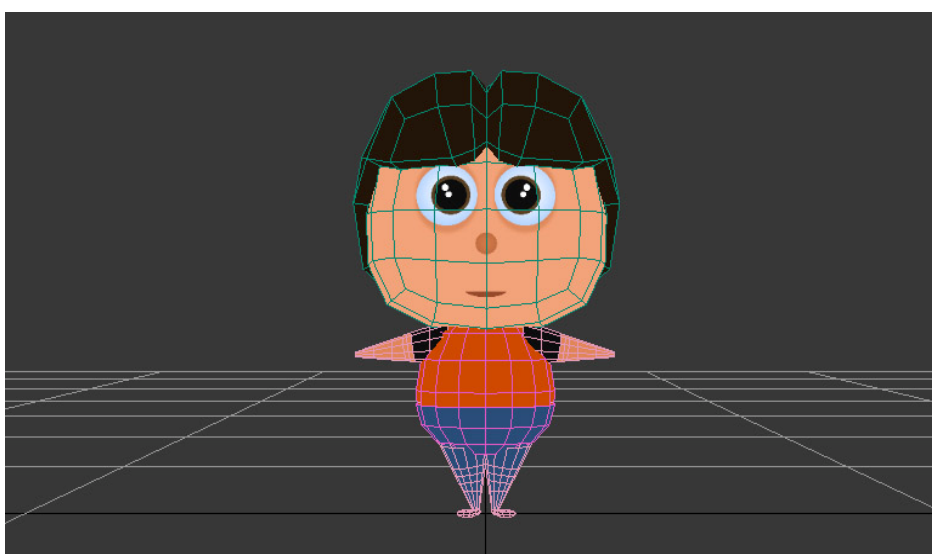


Fig. 4.55 Modellazione finale

Ossatura ed Animazione

Per dare vita al *Rigging* (ossatura) è stato utilizzato un plug-in di 3D studio Max disponibile nel pannello Create. Il plug-in in questione si chiama *Character Animation Tool (CAT)* ed è disponibile fin dalla versione 2010 del programma. Il CAT facilita il *Rigging* dei personaggi e la loro animazione. Per maggiori informazioni consultare l'indirizzo web: download.autodesk.com/esd/3dsMax/cat-help-2010/index.html.

Cliccando *Helpers* dal menu a tendina selezionare *CAT Objects*. In seguito selezionare *CATParent* ed infine utilizzare come *CATRig* il *Base Human*. Il *CATRig* consiste in una gerarchia di oggetti che definisce il sistema di movimento dell'ossatura del CAT. Una volta selezionato *Base Human* posizionarsi sulla *Viewport* e cliccare per inserire l'oggetto *Rig* mentre nei *CATRig Parameters* utilizzare *CATUnit Ratio* per modificare la sua scalatura in proporzione alla *mesh* del personaggio.

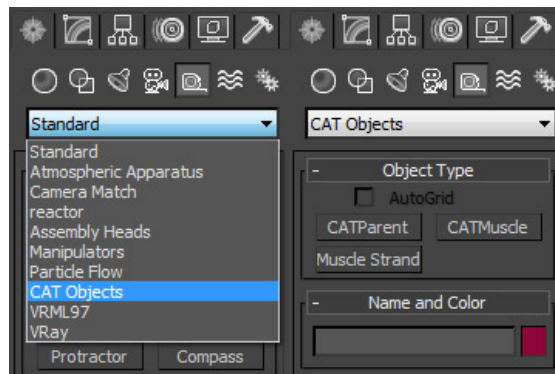


Fig. 4.56 Creazione del CAT Objects

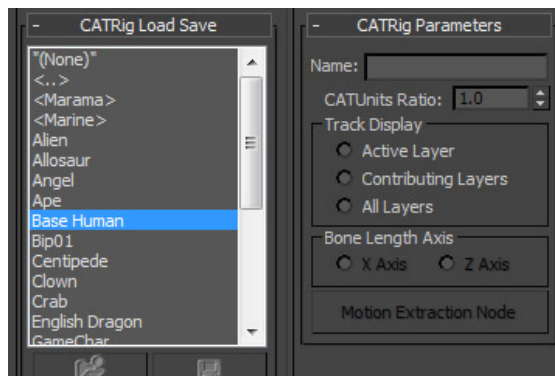


Fig. 4.57 Creazione del CATRig

Per agevolare la creazione dell'ossatura conviene rendere trasparente l'oggetto selezionando la voce *Object Properties* dal menù disponibile dal tasto destro del mouse e poi nel tab *General* diminuire *Visibility* nel pannello *Rendering Control*. Quindi aggiungendo il segno di spunta davanti alla opzione trasparenza oppure dal pannello screen spuntare l'opzione geometry nel pannello interno *Hide By Category*. Inoltre conviene congelare la pelle (per disabilitarne la selezione) sempre dal menù richiamabile dal tasto destro del mouse cliccando sull'opzione *Freeze Selection*. La pelle potrà essere nuovamente selezionata scongelandola sempre dal menù disponibile col tasto destro del mouse *Unfreeze All* (scongelatutto).

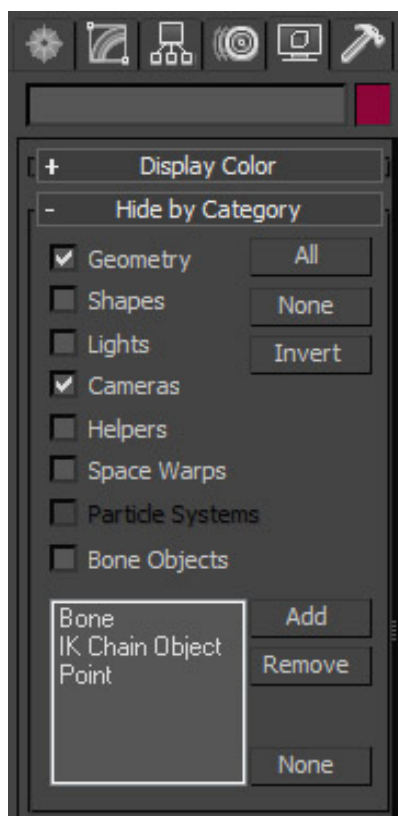


Fig. 4.58 Funzione Hide By Category del pannello Display

Dopo avere posizionato l'oggetto *CATRig* è necessario fare in modo che abbia la forma più simile possibile a quella del personaggio. Dovremmo ottenere una ossatura simile a quella in figura. In seguito affinché una pelle possa essere modificata da una ossatura questa deve essere agganciata ad uno scheletro.

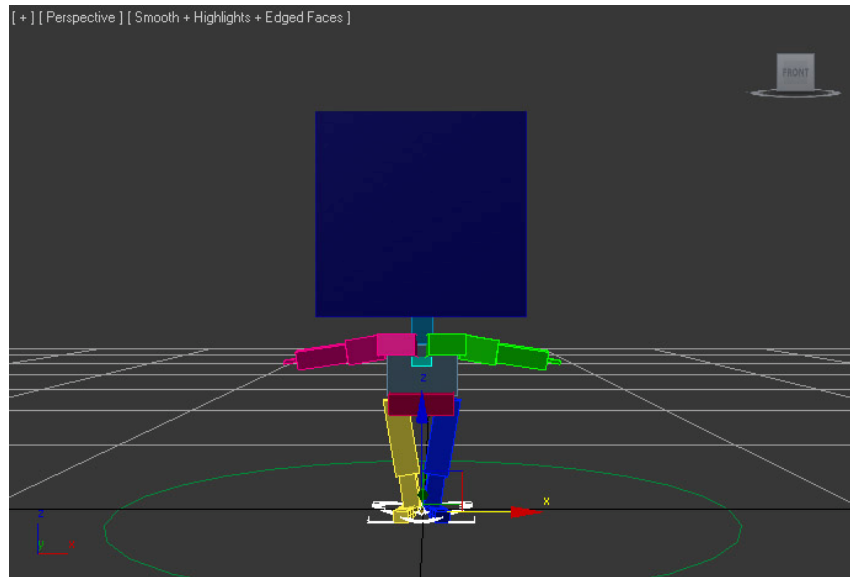


Fig. 4.59 Rigging del personaggio

Inoltre, occorre stabilire quanto ogni vertice della pelle sia influenzato dai giunti di una ossatura. Questa operazione si chiama pesatura della pelle. Una mesh diventa una pelle quando le si associa un modificatore *Skin*. Per prima cosa occorre selezionare quali giunti possono modificare la pelle premendo il pulsante aggiunti nel parametro del modificatore. Quindi si selezionano tutti i giunti necessari e si procede a pesare la pelle attraverso il comando *Edit Envelopes* (modifica involucri).

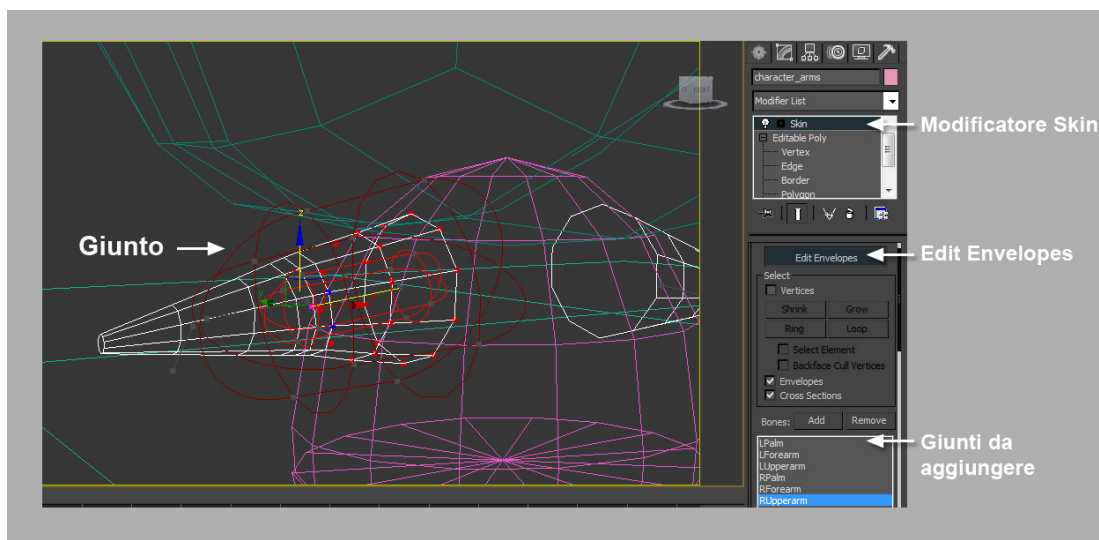


Fig. 4.60 Operazione di pesatura della pelle e giunti da aggiungere

3D Studio Max mette a disposizione due modalità distinte per pesare la pelle: la modalità automatica e la modalità manuale.

In modalità manuale è invece possibile specificare per ogni vertice quanto esso sia legato ad ogni singolo osso. In entrambi i casi si parte selezionando dall'apposito elenco l'osso a cui si vogliono associare i vertici.

Nella modalità automatica la pelle viene pesata in base alla distanza dalle ossa. I pesi dei vertici sono determinati a seconda di dove sono posizionati rispetto alle due capsule. I vertici contenuti all'interno della capsula più interna sono interamente mossi dall'osso considerato. Quelli contenuti nella capsula più interna sono solamente parzialmente influenzati dall'osso associato. L'entità dell'influenza dipende dalla distanza: più vicini sono alla capsula esterna meno sono legati all'osso considerato. I vertici al di fuori della capsula esterna non sono influenzati dall'osso selezionato. L'operazione di pesatura andrà effettuata sulle braccia, sulle gambe, sul corpo, sul bacino e sulla testa del personaggio.

Per applicare il movimento della ossatura insieme a tutta la *mesh* bisogna assegnare un nuovo livello di animazione dal pannello *Motion*. Cliccando sul pulsante *Abs* del pannello *Layer Manager* si potrà aggiungere un nuovo livello ed il tipo di animazione prescelta. Selezionare l'ultima opzione disponibile dal menu a discesa. L'opzione selezionata andrà ad aggiungere un nuovo livello chiamato *100% Cat Motion Layer* nel pannello *Layer Manager*.

Per poter creare l'animazione desiderata bisogna modificare i parametri interni al livello inserito poichè ogni singola parte della mesh statica ora è legata ad un osso dell'ossatura attraverso il processo appena visto detto anche *Skinning*. Grazie a questo processo la pelle del personaggio si muoverà in proporzione alla pesatura che è stata creata. Infine per modificare il movimento del personaggio premere l'icona con la forma a zampa di gatto. Si aprirà una finestra di dialogo dove è possibile cambiare i parametri di animazione di ogni singolo gruppo dell'ossatura per ottenere il tipo di camminata voluta.



Fig. 4.61 Pannello Layer Manager

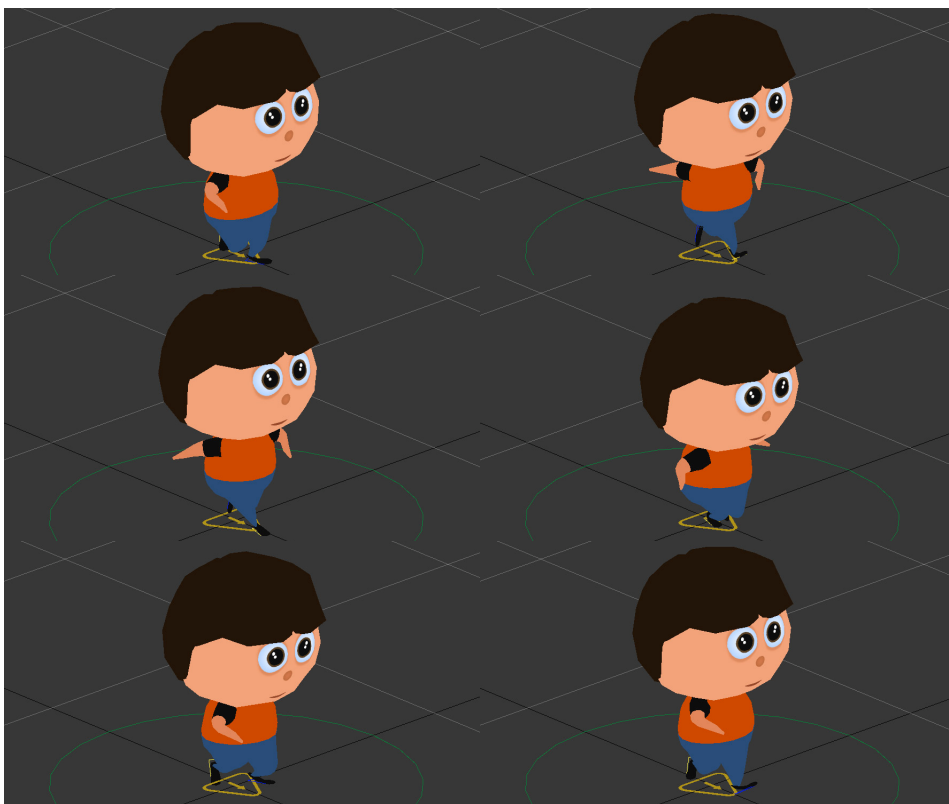


Fig. 4.62 Animazione finale del personaggio

4.3 Esportazione / Importazione del modello:

Dopo aver creato la scena in 3D Studio Max è possibile esportare i modelli selezionati in Unity nel formato *FBX*, usando il plugin fornito da Autodesk o usando altri formati generici. Unity può anche convertire i file “.max” nel formato *FBX*, ma per poter avviare questo processo, Unity richiede l’apertura di 3D Studio Max.

Creare un file in 3D Studio Max ed esportarlo può avere vantaggi o svantaggi, l’importazione prevede queste caratteristiche:

- tutti i nodi con posizione, rotazione e scalatura
- punti di pivot e nomi
- mesh con la normale e uno o due set di *UV Maps*
- materiali con texture di tipo diffusivo e colori e materiali multipli.
- animazioni e animazioni create attraverso l’ossatura

Per esportare manualmente nel formato *FBX* da 3D Studio Max è necessario scaricare l’ultima versione dell’esportatore di *FBX* dal sito web Autodesk e installarlo quindi procedere esportando la scena (*File -> Export o File -> Export Selected*) in formato *FBX*. Utilizzando le opzioni di esportazione di default dovrebbe essere a posto. Copiare il file esportato nella cartella *fbx* del progetto in Unity. Quando si torna in Unity, il file con estensione “.*fbx*” viene importato automaticamente. Quindi trascinare infine il file dalla finestra di progetto nella scena di gioco per poter visualizzare a schermo il risultato finale.

Opzioni di esportazione

Utilizzando le opzioni predefinite dell’esportatore di *FBX* (che, in fondo può esportare tutto ciò che è presente nel file) è possibile scegliere la seguente opzione:

- *Texture Embed* – questa opzione memorizza le mappe d’immagine nel file, ottimo per la portabilità, non così ottimizzato per la dimensione del file



Fig. 4.63 Impostazione di esportazione in 3D Studio MAX

Esportazione di animazioni Bone-based:

C'è una procedura da seguire quando si vogliono esportare ossa ed animazioni:

1. Impostare la struttura ossea a piacimento
2. Creare le animazioni che si desidera
3. Selezionare tutte le ossa e / o *IK solver*

4. Andare in *Motion* -> *Trajectories* e premere *Collapse*. Unity creerà un filtro chiave, quindi la quantità di keys che si esporta è irrilevante
5. "Esporta" o "Esporta selezionato" come nuovo formato FBX
6. Importare il file FBX negli Assets di Unity
7. In Unity è necessario riassegnare la texture del materiale alla radice dell'ossatura

Quando si esporta una gerarchia ossea con mesh e animazioni da 3D Studio Max a Unity, la gerarchia del *GameObject* prodotta corrisponde alla gerarchia che si può vedere nel pannello *Schematic View* in 3ds Max. Una differenza è che Unity creerà un *GameObject* come nuova radice, che conterrà le animazioni, e metterà la mesh e i materiali nella radice.

Se si preferisce avere informazioni animazioni e mesh nello stesso *GameObject*, passare alla *Hierarchy View* in 3ds Max, e parentare la mesh nella gerarchia delle ossa.

Esportazione di due set UV per Lightmapping:

Il “*Render To Texture*” di 3D Studio Max e le funzionalità di *unwrapping* automatico possono essere utilizzate per creare *lightmap*. Si noti che Unity ha un motore interno per creare le *lightmap*, ma si potrebbe preferire usare 3D Studio Max se questo si adatta meglio al vostro flusso di lavoro. Solitamente un set di *UV Map* viene utilizzata per le texture principali e / o per le *normal map*, e un altro set di *UV Map* viene utilizzata per la texture delle *lightmap*. In entrambe le UV, per essere impostate correttamente, il materiale in 3D Studio Max deve essere *Standard* e *Diffuse* (per la texture principale) e *Self-Illumination* (per le *lightmap*). Gli *slot* delle mappe devono essere impostati come da figura seguente.

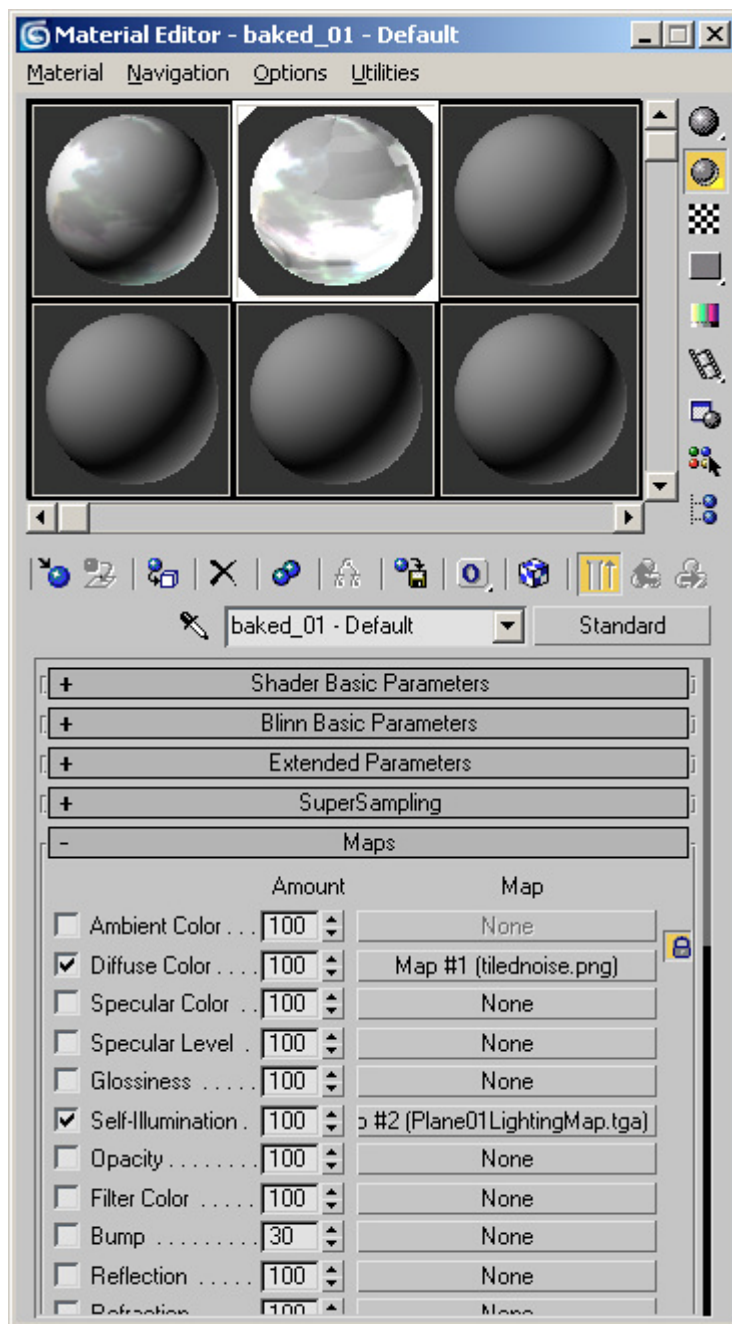


Fig. 4.64 Impostazioni del pannello Materiali per il Lightmapping in 3ds Max, usando una mappa di tipo Self-Illumination.

4.4 UV Mapping e Texturing

4.4.1 UV Mapping

L'UV Mapping è il processo di modellazione 3D che consiste nel realizzare un'immagine 2D a partire da un modello 3D.

Questo processo proietta una mappatura della texture su un oggetto 3D. Le lettere "U" e "V" indicano gli assi della texture 2D, poichè "X", "Y" e "Z" sono già utilizzati per indicare gli assi dell'oggetto 3D nello spazio del modello.

Una mappa UV è una texture associata a facce specifiche della mesh, creando una corrispondenza tra punti dell'immagine – secondo il sistema di riferimento bidimensionale UV, equivalente al cartesiano XY – e vertici della mesh, opportunamente “scuciti” e disposti sul piano UV.

L'UV Texturing permette ai poligoni che compongono un oggetto 3D di essere disegnati con i colori di un'immagine. L'immagine in questione si chiama UV texture map (quando usiamo i quaternioni anche la coordinata W viene usata infatti si parla di UVW mapping) ma è solo un'immagine 2D niente di più. Il processo di mappatura UV comporta l'assegnazione di pixel nell'immagine per mappare la superficie del poligono, che solitamente viene fatta "programmaticamente" copiando un pezzo a forma di triangolo dalla mappa dell'immagine e incollandolo su un triangolo per l'oggetto. [2] Quini l'UV è l'alternativa al XY, si occupa solo di mappare nella spazio delle texture piuttosto che nello spazio geometrico dell'oggetto. Ma il calcolo del rendering utilizza le coordinate UV per determinare come dipingere la superficie tridimensionale.

Quando un modello viene creato come una mesh poligonale utilizzando un software di modellazione 3D, le coordinate UV possono essere generate per ogni vertice della mesh. Un modo per il modellatore 3D di aprire la mesh triangolare lungo le cuciture, è quello di disporre i triangoli su una pagina piatta. Se la mesh è una sfera UV, per esempio, il modellatore la può trasformare in una proiezione equirettangolare. Una volta che il modello è stato “aperto”, l'artista può dipingere una texture su ogni triangolo individualmente, utilizzando la *mesh* mappata sull'immagine rettangolare come modello. Quando la scena verrà renderizzata,

ogni triangolo utilizzerà la sua corrispondente texture ottenuta dalla mappatura precedentemente creata per texturizzare la superficie corrispondente.

Il processo di mappatura UV nel modo più semplice richiede tre passaggi: *unwrap* della *mesh*, creare la texture, e applicare la texture.

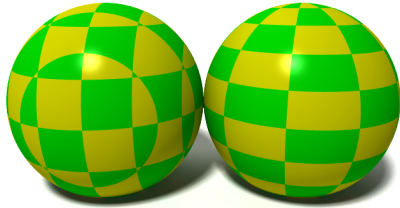


Fig. 4.65 Una sfera, senza e con UV mapping.

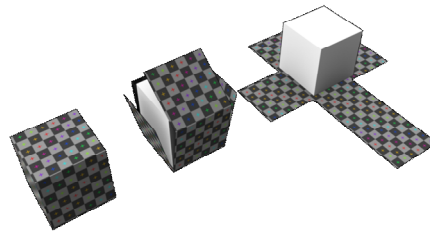


Fig. 4.66 Una rappresentazione dell'UV mapping di un cubo.

4.4.2 Texturing

Vedremo ora come utilizzare la tecnica di mappatura UVW in 3DS Max, al fine di essere in grado di applicare texture complesse per i modelli creati. La tecnica di mappatura UVW prevede la creazione di una 'mappa del modello' che poi bisogna colorare e “cucire” di nuovo sul modello come una texture. In questo modo sarà possibile applicare la texture utilizzando una singola mappa, invece di crearne più di una per le varie sezioni del modello e questo è il metodo che di solito è utilizzato per la creazione di texture nei videogiochi.

Una volta creato il modello, dopo averlo selezionato, accedere al pannello *Modify* (modifica) e applicare *UVW Unwrap* dalla lista dei modificatori. Nell'elenco degli *Stack* del pannello modifica, fare clic sul segno “+” accanto al modificatore UVW Unwrap per svelare un elenco di tre opzioni. Le tre opzioni disponibili (*Vertex*, *Edge*, e *Face*) rendono possibile selezionare un gruppo o un singolo vertice, un bordo o una faccia dalla viewport. Utilizzare l'opzione più comune ovvero *Face*, quindi per andare avanti selezionarla. Una volta selezionata questa opzione sarà possibile cliccare sui poligoni del modello che interessano per potere poi applicare altri comandi.

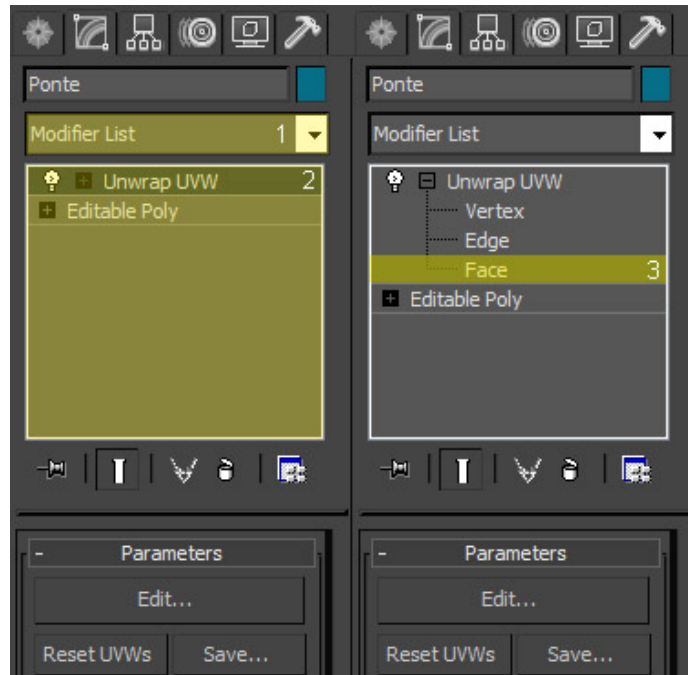


Fig. 4.67 Applicazione Modificatore Unwrap UVW

Tornare alla scena e selezionare tutte le facce che compongono il tetto. Sarà necessario tenere premuto il tasto *Ctrl* sulla tastiera mentre si procede per essere in grado di selezionare più oggetti contemporaneamente. Una volta fatto, cercare il menù a discesa *Parameters* e fare clic su *Edit*, a questo punto si aprirà la finestra *Edit UVWs*. Potrebbe non essere possibile riconoscere la figura rossa ma questa è la mappatura UV iniziale da modificare.

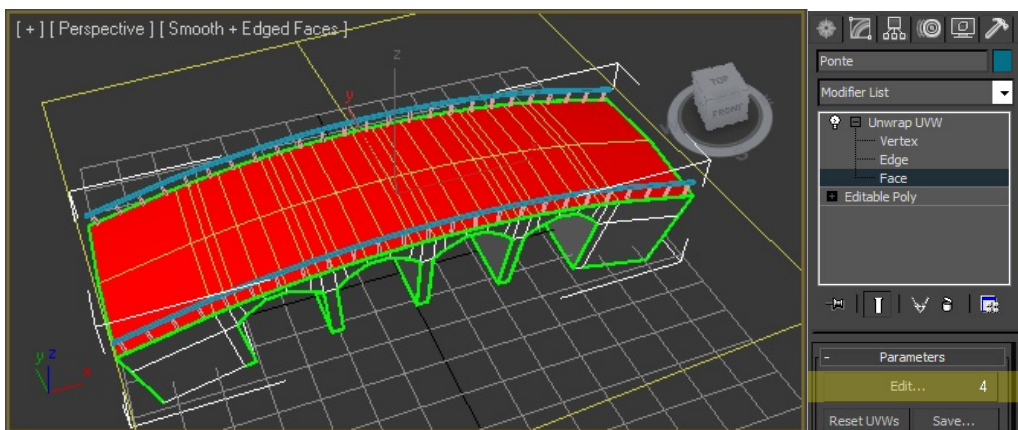


Fig. 4.68 Modificatore Unwrap UVW

Mentre la finestra *Edit UVWs* è ancora aperta, accedere al pannello *Modifica*, scorrere fino a *Map Parameters*, quindi premere su *Planar* che dovrebbe cambiare

la mappatura UV sulla finestra *Edit UVWs*. Infine, assicurarsi di deselezionare l'opzione di normalizzazione prima di passare alla fase successiva.

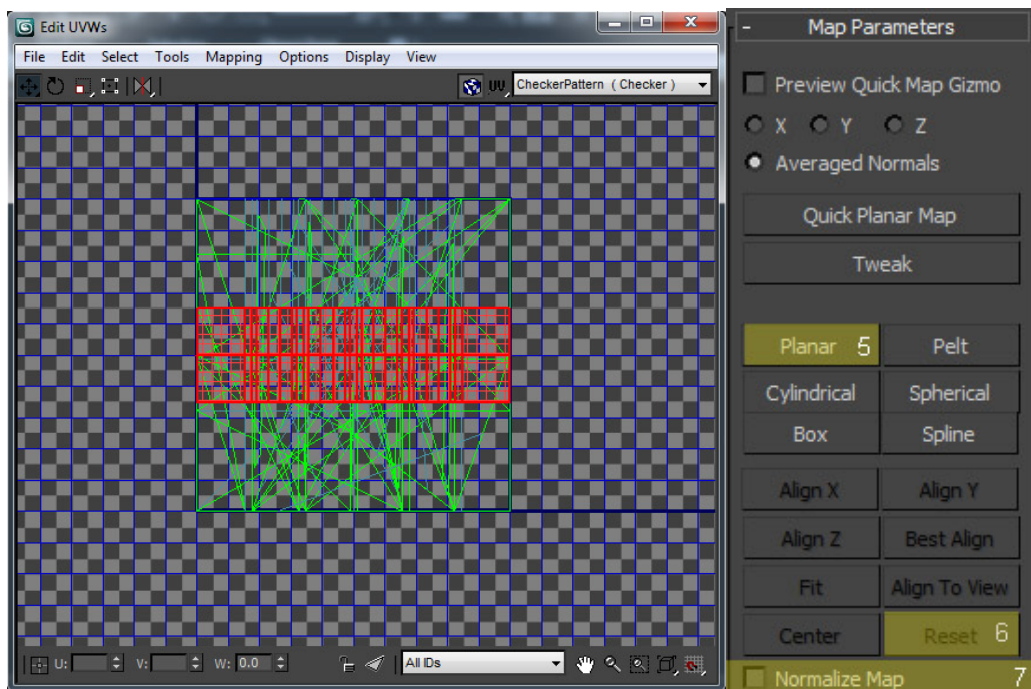


Fig. 4.69 Finestra Edit UVW

Premere il pulsante di *Planar*, ancora una volta per disattivarlo. A questo punto sarà possibile spostare la UV map delle facce selezionate da parte. Potrebbe essere conveniente scalare l'immagine usando una scala di 1:1 in modo da poter ottenere un workspace più grande e per poter lavorare con le altre UV map che si andranno a creare di seguito. Si consiglia di utilizzare il metodo appena visto per fare l'unwrap del resto del modello utilizzando le seguenti opzioni del pannello Map Parameters: *Planar*, *Pelt*, *Cylindrical*, *Spherical* o *Box*.

Ecco un altro breve riassunto su come fare. Selezionare le facce che si desiderano mappare, ad esempio le pareti frontali. Utilizzare uno dei parametri di mappatura, potrebbe anche essere necessario utilizzare uno degli strumenti di allineamento ad esempio *Align X*, *Align Y*, *Align Z*. Disattivare il parametro appena usato e quindi spostare o ridimensionare la mappatura UV e sposterla da qualche parte accanto alla mappa UV precedente. Ripetere l'operazione per mappare tutte le facce del modello.

Alla fine di tutto il processo si dovrebbe ottenere una mappa simile a quella qui sotto. Il passo successivo è quello di portarle dentro un editor di immagini per disegnare le texture reali. Infine renderizzare le mappe UV per ottenere un'immagine. Cliccare nel menu *Tools > Render UVW Template*. Si aprirà la finestra *Render UVs*, modificare la larghezza e l'altezza a 1024 pixel e premere il pulsante *Render UV Template*. Una volta che il rendering è completo, salvare il file e aprirlo con un editor di immagini.

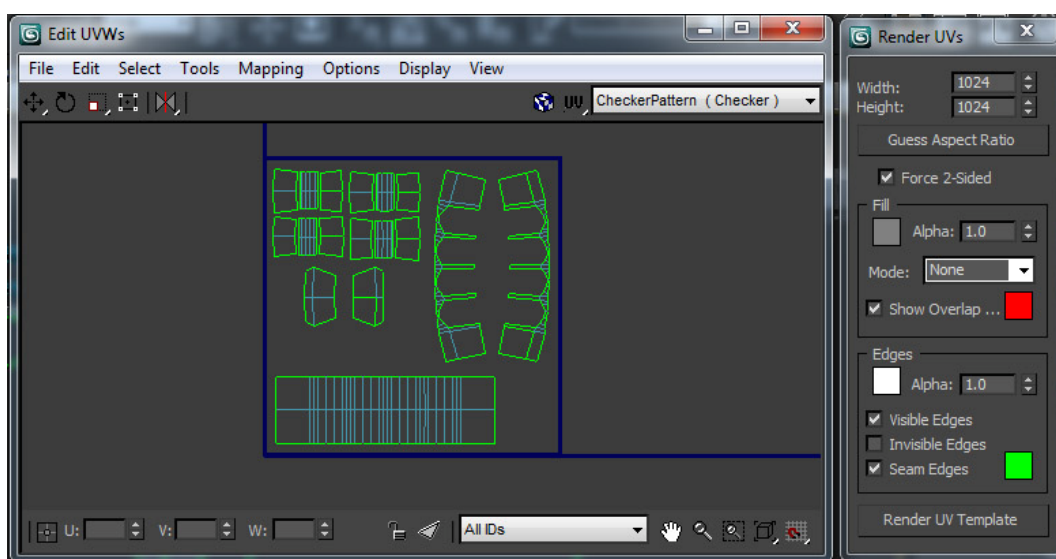


Fig. 4.70 Edit UVWs: editor per la creazione della mappa UV.

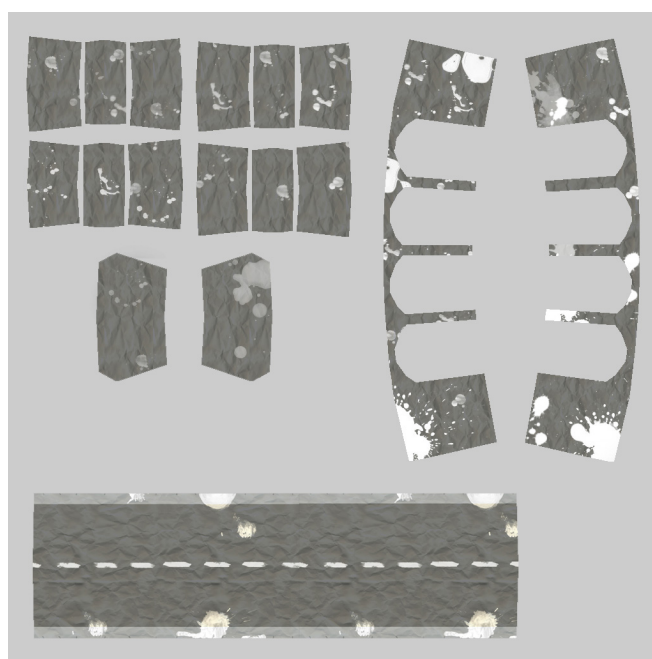


Fig. 4.71 Texture map dopo l'editing in Photoshop.

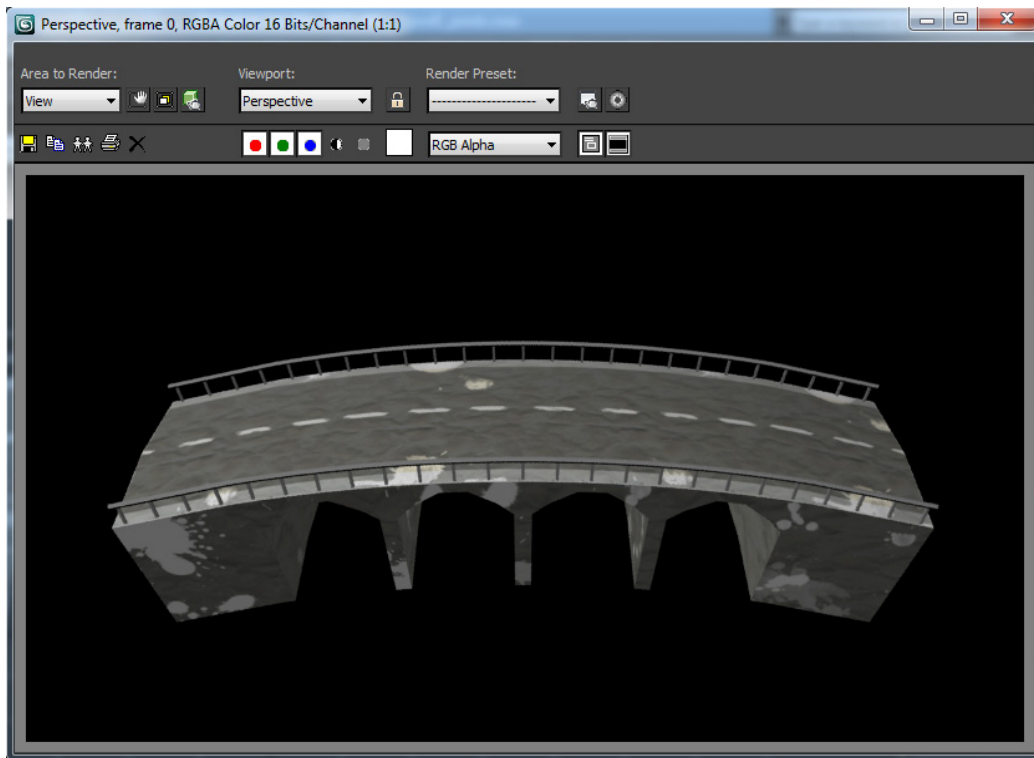


Fig. 4.72 Rendering finale del ponte in 3ds Max.

4.5 Render to Texture e LightMapping

Creare ombre in tempo reale può incidere notevolmente sulle performance del videogioco inoltre può dare risultati poco soddisfacenti. Proprio per questo motivo per avere un'estetica migliore e per dare al gioco una maggiore fluidità si è fatto uso delle lightmap ottenute attraverso la tecnica del *Render to Texture* direttamente da 3D Studio Max. Grazie all'uso delle *lightmap* è stato possibile aggiungere una maggior lucentezza alle superfici e una miglior resa delle ombre su oggetti statici come edifici e case oltre che a risparmiare in termini di calcolo sulle varie superfici illuminate.

Una lightmap è una struttura dati che contiene informazioni sulla luminosità delle superfici in applicazioni di grafica 3D come i videogiochi. Le lightmap sono pre-calcolate, e normalmente vengono utilizzate per gli oggetti statici. Sono particolarmente adatte per ambienti urbani ed interni con grandi superfici planari. Quando si crea una lightmap qualsiasi modello di illuminazione può essere utilizzato perchè l'illuminazione viene interamente pre-calcolata. Tra le varie

tecniche usate troviamo *l'Ambient Occlusion*, l'illuminazione diretta con ombre pre-campionate e tutte le soluzioni di *Radiosity* con luce di rimbalzo.

I moderni software di modellazione includono plugin specifici per l'applicazione delle *lightmap* nelle coordinate UV, per la disposizione di più superfici in un'unica immagine mappata poi da una texture (tecnica dell'unwrap) e per il *rendering* finale delle stesse mappature. In alternativa i game engine possono includere strumenti interni per la creazione delle *lightmap* come accade ad esempio in Unity con il plugin interno Beast.

Prima che le *lightmap* fossero inventate le applicazioni in tempo reale erano basate esclusivamente sulla tecnica del *Gouraud Shading* per interpolare l'illuminazione dei vertici sulle superfici. Ciò permetteva solo informazioni a bassa frequenza sull'illuminazione e talvolta fenomeni di clipping sulla camera senza un'interpolazione corretta della prospettiva [1].

Per creare le *lightmap* è necessario partire dal modello del ponte visto nel precedente capitolo. Si dovrà applicare la tecnica del Render to Texture alla mappatura creata attraverso il modificatore *UVW Unwrap*. Prima di tutto salvare la mappatura creata in precedenza in un file con estensione “.uvw” e poi inserire un nuovo modificatore *Unwrap UVW*. Cambiare il canale inserendo il valore “2” nel pannello Parameters del modificatore. In seguito caricare la mappatura appena salvata.

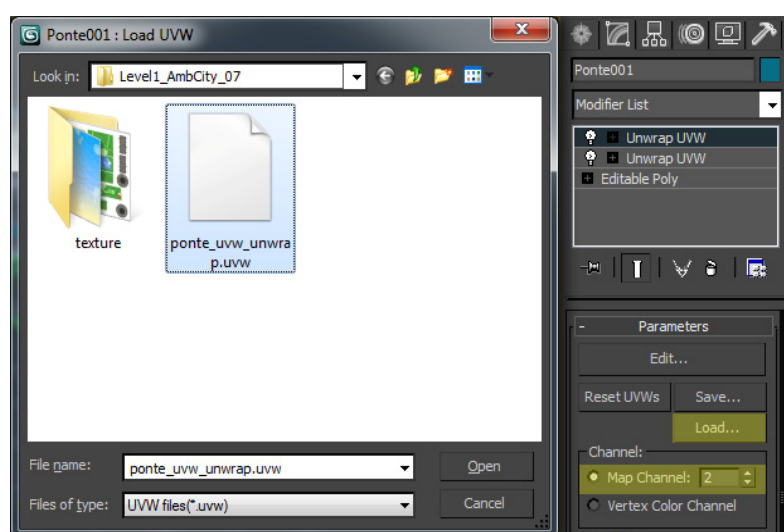


Fig. 4.73 Caricamento della UVW

Premere il tasto (0) per aprire la finestra relativa al Render to Texture. Selezionare *Use Existing Channel*, cambiare il canale a "2", scegliere *Add* e poi *Lightingmap* come nell'immagine sottostante.

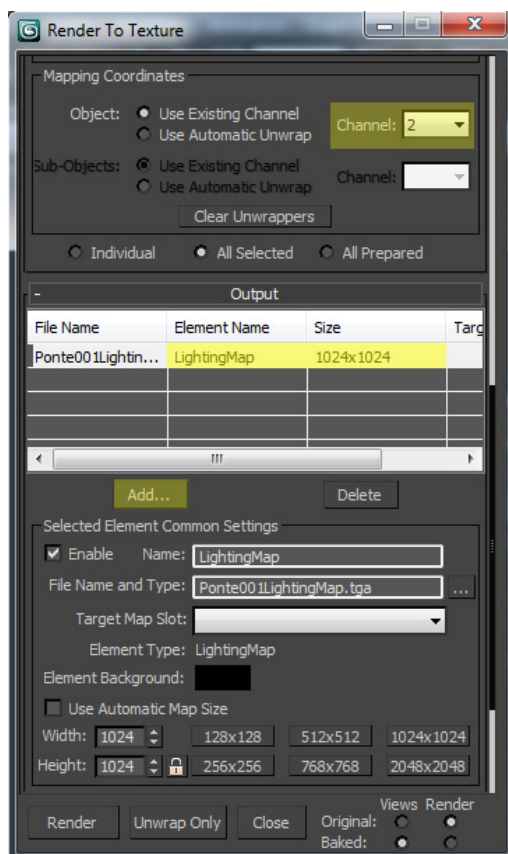


Fig. 4.74 Pannello Render To Texture

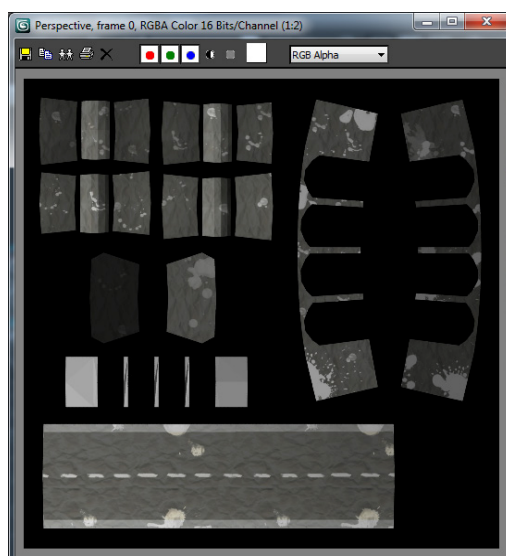


Fig. 4.75 Rendering della texture

L'immagine che si vede, ottenuta dal rendering, non è il risultato reale. L'immagine reale è stata memorizzata nella cartella di 3ds Max, di solito viene memorizzata nella cartella "documents/3dsMax/sceneassets/images". Una volta situati in questa cartella cercare il file composto dal nome del modello unito a "LightingMap" e con estensione ".tga". Il file corrispondente dovrebbe risultare come quello visualizzato nella figura sottostante.

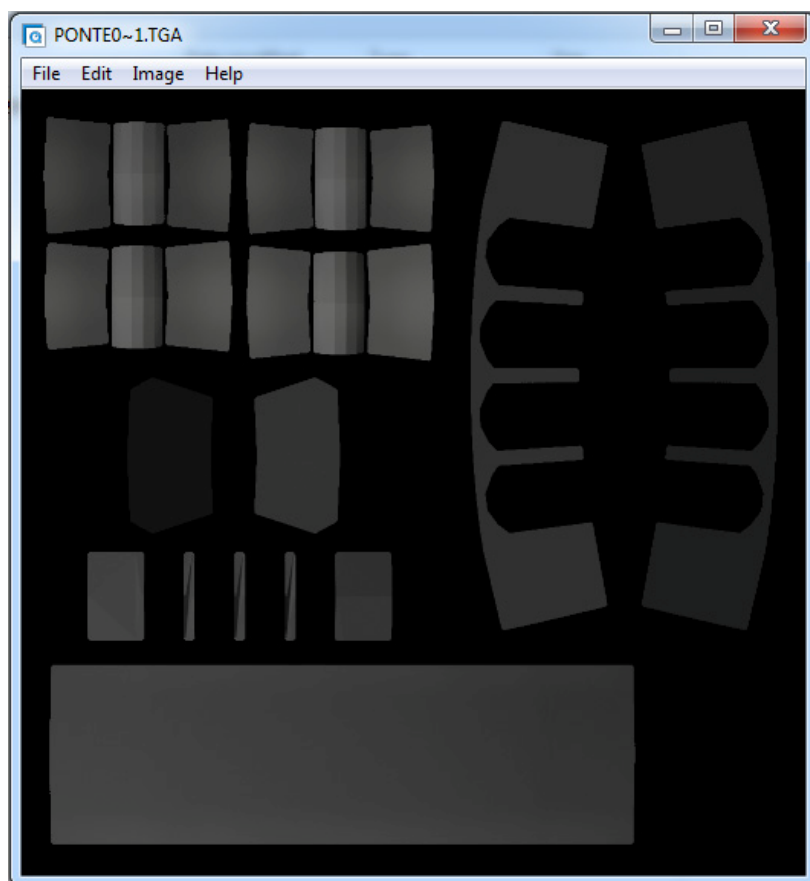


Fig. 4.76 PonteLightingMap.tga

A questo punto per poter proseguire ed aggiungere la lightmap al modello il modo più semplice è quello di usare la funzione *Pick Material from Object* e quindi copiare il *Baked Material* (materiale creato) in uno slot vuoto poichè il plugin per esportare in formato FBX attualmente non supporta gli "*Shell Material*". Una volta creato il nuovo materiale è necessario aggiungere la nuova bitmap "PonteLightingMap.jpg" creata attraverso il *Render to Texture* nel parametro *Diffuse* del pannello *Basic Parameters* che si trova dentro la finestra *Material Editor*.

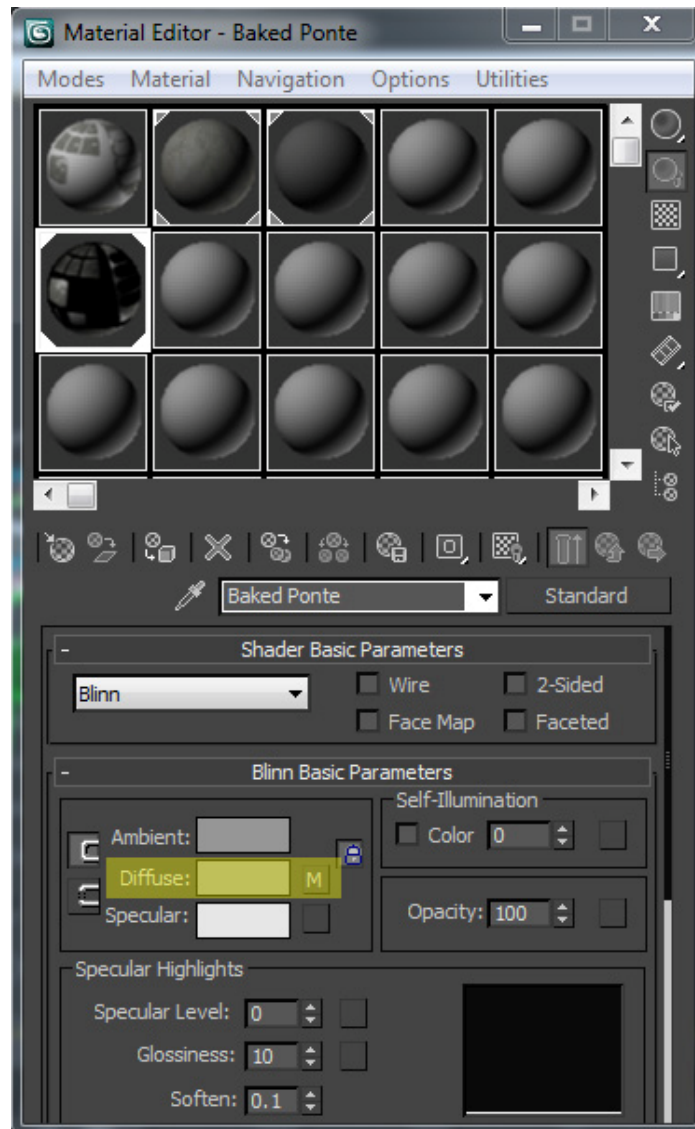


Fig. 4.77 Creazione del nuovo materiale attraverso la lightmap texture

Dopo aver assegnato il materiale al modello si può finalmente esportare il modello con l'estensione “.fbx” attraverso il plugin di 3D Studio Max. Nelle varie opzioni di esportazione ricordarsi di spuntare *Embed Media* in modo da poter importare in Unity anche le texture usate.

Infine, una volta aperto Unity ed importato il file FBX appena creato, attraverso il menù *Assets -> Import New Asset* è possibile vedere che oltre alla mesh del modello verrà importata anche la texture inclusa. Solitamente Unity colloca le texture importate in una cartella nominata in questo modo: “nome del modello” + “.fbm”.

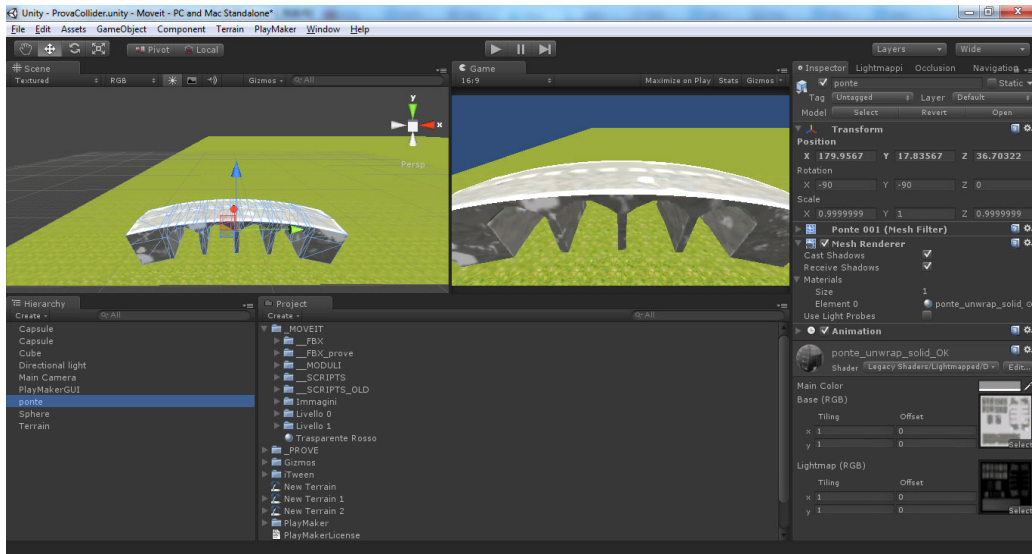


Fig. 4.78 Effetto finale dopo l'importazione in Unity

Capitolo 5

Sviluppo

Nel capitolo sullo sviluppo vengono prese in esame le *state machine* usate e tutti i principali script per implementare i comportamenti del personaggio, degli oggetti e dell'interfaccia grafica. Le state machine permettono di aggiungere azioni ed eventi in maniera visuale, proprio per questo si è scelto di usare PlayMaker per implementare in maniera più diretta alcuni comportamenti base e per gestire l'invio e la ricezione di eventi tra i vari *GameObject*. Per implementare gli algoritmi più avanzati, come ad esempio la randomizzazione dei nemici e l'interfaccia grafica, si è scelto invece di utilizzare invece il linguaggio Javascript per avere una maggior padronanza del codice.

5.1 State Machine

5.1.1 Personaggio

Il personaggio principale del gioco fa uso di due *Finished State Machine* (FSM) di PlayMaker. Una chiamata "TriggerManager" per gestire l'evento relativo e l'altro chiamata "MoveManager" per gestire l'attivazione o la disattivazione dei movimenti dopo le collisioni. Qui di seguito si elencheranno le funzionalità di ogni FSM.

Trigger Manager

Questa FSM si occupa principalmente di gestire l'evento di sistema *TRIGGER ENTER* ogni volta che il personaggio penetra uno dei *Collider* presenti negli oggetti sulla scena. Questa *state machine* si compone di due stati: "Attivo" e "TriggerEnter". Il primo è lo stato di partenza e al suo interno non vi è presente alcuna azione. L'unico evento che può lanciare è quello di sistema *TRIGGER*

ENTER. Una volta lanciato l'evento la *state machine* entra nello stato "TriggerEnter". In questo stato vengono avviate due azioni: *Get Trigger Info* e *Send Event By Name*.

La prima azione si occupa di capire quale oggetto sia stato colpito dal personaggio e poi salva queste informazioni in una variabile globale chiamata "ColliderObject". La variabile "ColliderObject" è fondamentale per il funzionamento di tutto il gioco. Infatti tramite quest'ultima è possibile inviare informazioni e veicolare un certo tipo di evento all'interno delle altre *state machine*.

La seconda azione invece si occupa di inviare l'evento "TriggerEnter" alla *state machine* presente all'interno dell'oggetto "ModuliManager". Una volta eseguite queste azioni la FSM torna nello stato "Attivo" in attesa di un altro evento *TRIGGER ENTER*.

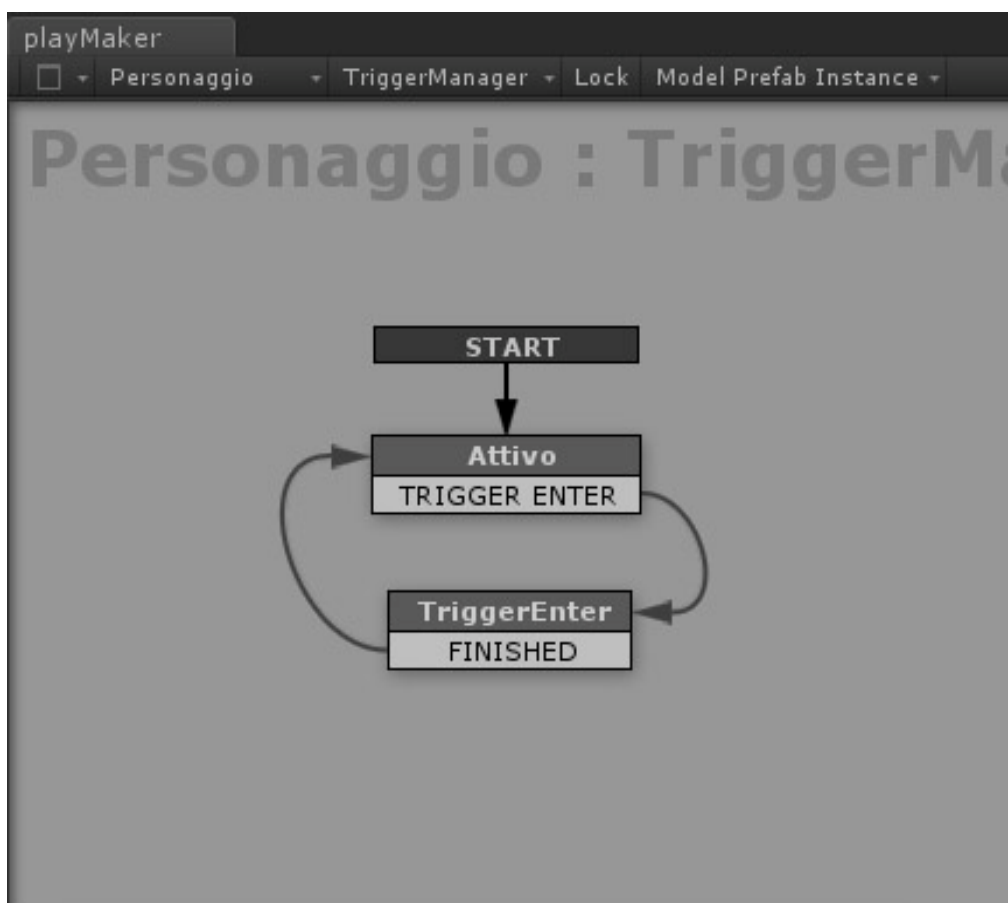


Fig. 5.1 La FSM TriggerManager

Move Manager

In questa FSM viene attivato / disattivato il movimento e l'interazione col personaggio. L'attivazione / disattivazione viene effettuata in base al funzionamento della FSM presente nel GameObject "MODULI_WRAPPER". Infatti in base all'evento lanciato da quest'ultima sarà regolato il comportamento della "MoveManager".

Nello stato di partenza (Attivo) sono diverse le azioni che servono a dare vita all'interazione col personaggio: *Play Animation*, *Set Property speed Personaggio*, *Set Property assey*, *Set Property traslazione* e *Set Property salto*. Ognuna di queste azioni ha il compito di avviare i comportamenti relativi.

Nel secondo stato (Passivo), attivabile attraverso l'invio dell'evento "Disattiva", verranno invece disattivati tutti i comportamenti appena elencati per un lasso di tempo relativo pari a quello trascorso per inviare l'evento "Attiva" da parte della FSM presente in "MODULI_WRAPPER". Una volta che "MODULI_WRAPPER" avrà inviato l'evento "Attiva" la FSM tornerà nello stato di partenza.

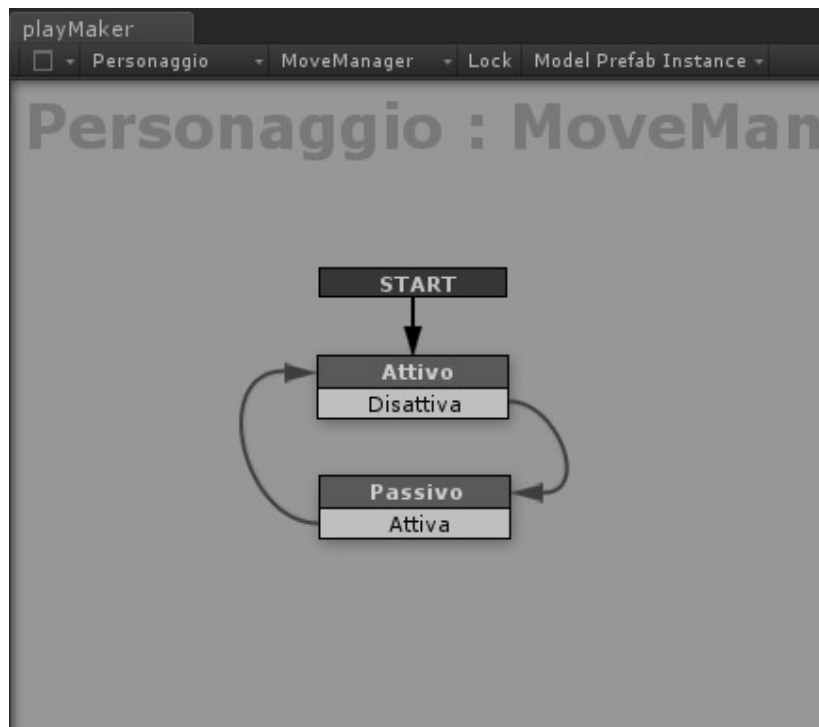


Fig. 5.2 La FSM MoveManager

5.1.2 Collider Manager

Il “Collider Manager” si occupa di gestire le collisioni. In base al nome dell'evento che gli sarà inviato lancerà eventi di tipo diverso che corrisponderanno a quello che dovrà accadere in base all'oggetto che ha colliso col personaggio.

Lo stato di partenza (Listener) non ha alcun azione e resta solo in ascolto dell'evento “TriggerEnter” che gli verrà inviato dal personaggio. Dopo l'invio dell'evento la State Machine entrerà nello stato “TriggerEnter”.

In questo stato abbiamo le seguenti azioni: *Send Event To Object* e quattro azioni di tipo *Game Object Compare Tag*. La prima azione si occupa di inviare l'evento “Collisione” all'oggetto “ColliderObject” (oggetto della collisione) mentre le altre quattro azioni inviano eventi in base al Tag che ha l'oggetto.

Esistono tre casi diversi: nel primo il “ColliderObject” ha un tag “Enemy” o “Building” ed invierà l'evento “TriggerEnterMalus”; nel secondo ha un tag “Bonus” ed invierà l'evento “TriggerEnterBonus”; nell'ultimo caso ha un tag “Checkpoint” ed invierà l'evento “TriggerEnterCheck”.

Se l'oggetto colliso è un nemico o un edificio la *state machine* entrerà in nuovo stato chiamato “EnterMalus”. In questo stato vi sono due azioni: *Send Event to Moduli Wrapper* e *Send Event to Cielo*. La prima si occuperà di mandare l'evento “Stop” al GameObject “MODULI_WRAPPER” mentre la seconda azione manderà l'evento “Stop” al GameObject “Cielo”.

Se l'oggetto colliso è un bonus la *state machine* entrerà nello stato “EnterBonus” nel quale l'unica azione è quella di inviare l'evento “Aggiorna Bonus” alla FSM interna a “GUIManager” in modo da aumentare il punteggio.

Infine se l'oggetto colliso è un *checkpoint* lo stato finale sarà “EnterCheckPoint”. Questo stato ha due azioni: *Send Event to Moduli Wrapper* e *Send Event to GUI Manager*. La prima azione invierà l'evento “Aggiungi Modulo” a “ModuliManager” per poter inserire un nuovo modulo al suo interno mentre la seconda azione invierà l'evento “Aggiorna Punteggio” all'oggetto “GUIManager”.

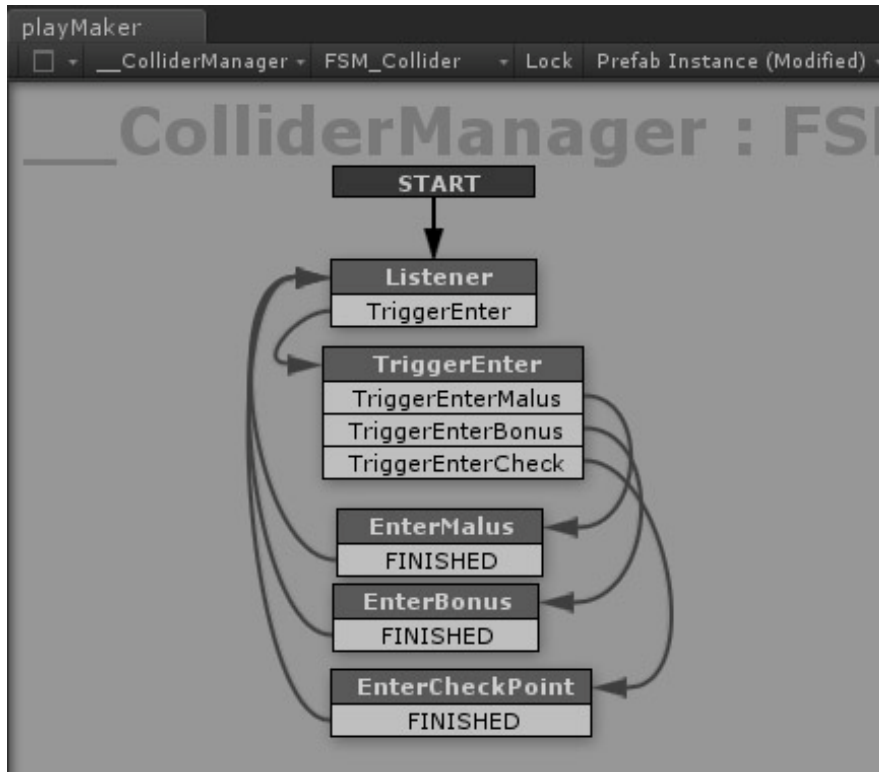


Fig. 5.3 La FSM ColliderManager

5.1.3 GUI Manager

La FSM presente in “GUIManager” gestisce l’aggiornamento del punteggio relativo al numero di *checkpoint* passati e del numero di *bonus* colpiti dal personaggio. Questa *state machine* ha uno stato di partenza (Inizializzazione) che si occupa di inizializzare le variabili interne.

Il secondo stato (Listener) rimane in ascolto degli eventi “Aggiorna Punteggio” ed “Aggiorna Bonus” che gli verranno inviati da “ColliderManager”. Nel caso sia lanciato il primo evento la FSM entrerà nello stato “Checkpoint”, mentre tramite il secondo entrerà nello stato “Bonus”.

Nello stato “Checkpoint” abbiamo quattro azioni: *Int Add*, *Convert Int To String*, *Build String* e *Set GUI Score*. Le prime due azioni si occupano di recuperare la variabile “CheckPointPassed” incrementando il valore di un’unità e di convertirla a stringa. La terza azione costruisce una stringa composta da due parti: “percorso” + “numero checkpoint”. L’ultima azione mostra il valore sull’interfaccia grafica grazie al GameObject “GUI_Score”.

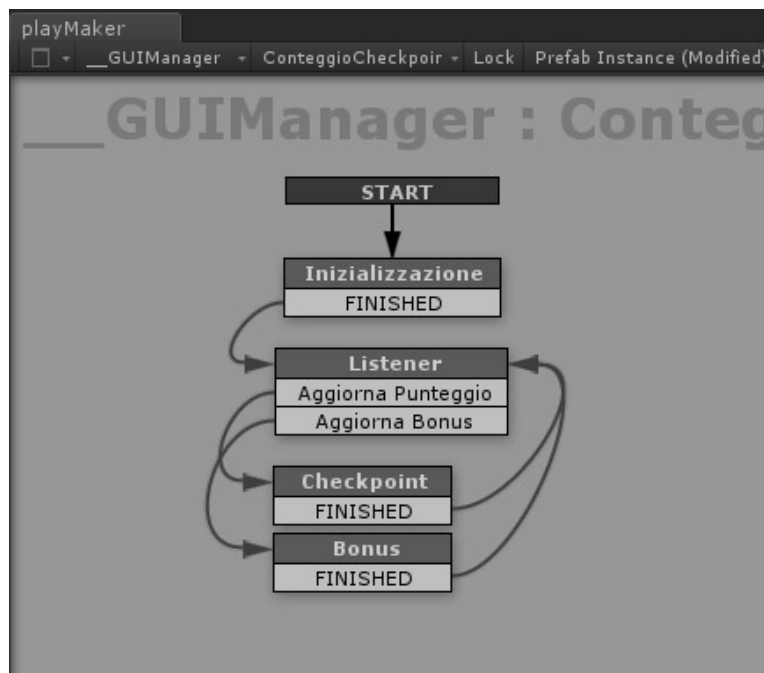


Fig. 5.4 La FSM GUIManager

5.1.4 Moduli Manager

Moduli Seguenti

Questa FSM si occupa di istanziare i moduli del gioco sulla scena e di posizionarli.

Lo stato di partenza (Listener) rimane in ascolto dell'evento "Aggiungi Modulo" che gli verrà inviato da "ModuliManager".

Il secondo stato (SceltaTipoModuli) ha un'azione: *String Switch*. Questo tipo d'azione seleziona il tipo di modulo da caricare attraverso la variabile globale "moduliDaCaricareGlobale" di tipo *String* che viene settata nel primo livello "start". In base alla stringa presente nella variabile verrà inviato l'evento "aggiungi campagna", "aggiungi periferia" oppure "aggiungi città".

Negli stati successivi viene selezionato un modulo a random dalla lista dei *Prefab* esistenti relativi alla scena settata dalla variabile globale. In seguito viene creato ed istanziato il modulo nello *Spawn Point* "CentroModuli" con la giusta rotazione e poi viene imparentato con "MODULI_WRAPPER" per ereditare la sua rotazione. Infine viene inviato l'evento "Controllo" all'oggetto *checkpoint* interno al modulo in modo che possa distruggere quest'ultimo dopo una certa rotazione.

L'ultimo stato (Posizionamento Moduli) si occupa di impostare posizione e rotazione del modulo e infine di aggiungere una rotazione di 60 gradi all'oggetto "CentroModuli". In questo modo il prossimo modulo che verrà caricato sarà ruotato di 60 gradi in più rispetto a quello attuale in cui si sta svolgendo la scena di gioco. Questo processo serve ad avere un caricamento continuo dei moduli della scena dando l'impressione che il personaggio avanzi lungo un percorso infinito mentre in realtà sono i moduli che ruotano continuamente.

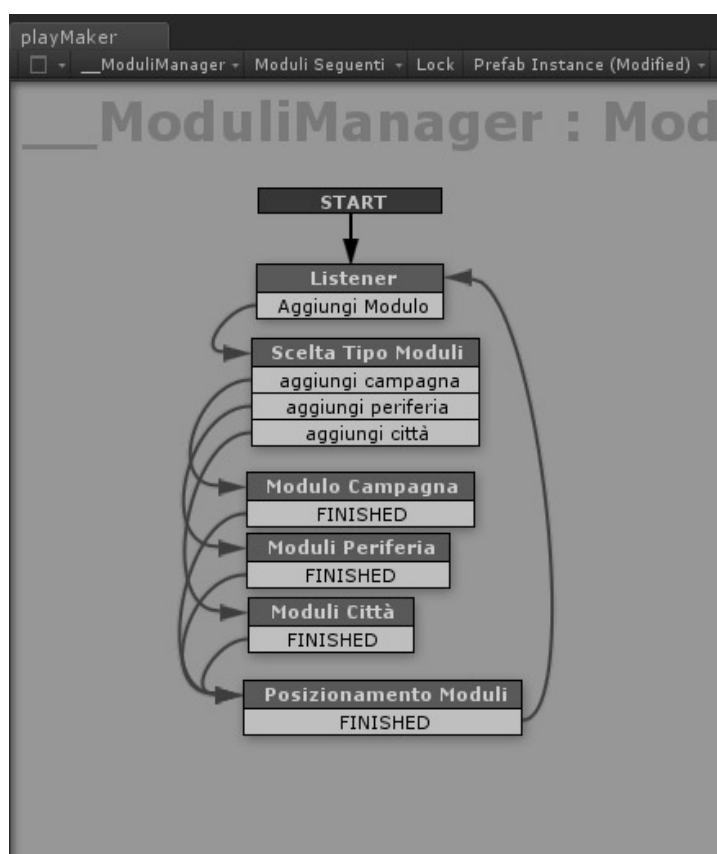


Fig. 5.5 La FSM ModuliManager

5.1.5 Nemici Manager

La FSM interna all'oggetto "NemiciManager" serve a creare i nemici del gioco sulla scena e di salvarli in una variabile "nemico" che viene usata dallo script "RandomObject.js" per creare gli ostacoli a random.

Lo stato di partenza (Scelta Tipo Nemici) ha un'azione: *String Switch*. Questo tipo d'azione seleziona il tipo di nemico da caricare attraverso la variabile globale "moduliDaCaricareGlobale" di tipo String che viene settata nel primo livello

“start”. In base alla stringa presente nella variabile verrà inviato l’evento “aggiungi campagna”, “aggiungi periferia” oppure “aggiungi città”.

Negli stati successivi (Campagna – Periferia – Città) vi è una sola azione: *Select Random Game Object*. Quest’azione seleziona a random uno dei GameObject di tipo “nemico” dalla lista degli oggetti che vanno impostati nell’azione. La lista degli oggetti elencati sono *Prefab* relativi alla scena settata dalla variabile globale. In seguito l’oggetto scelto viene salvato nella variabile “nemico” che verrà usata dallo script “RandomObject.js”.

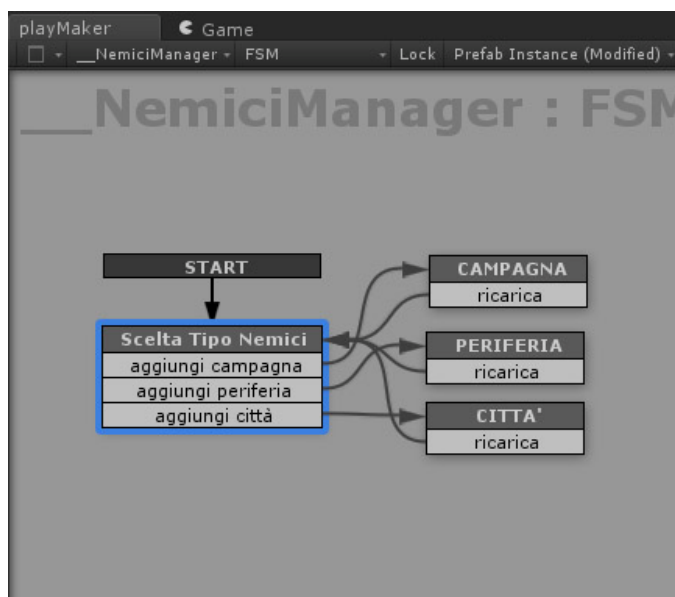


Fig. 5.6 La FSM NemiciManager

5.1.6 Moduli Wrapper (Contenitore dei Moduli)

La FSM di questo *GameObject* si occupa di gestire la rotazione di tutti i moduli presenti nella scena in base alle collisioni avvenute con gli oggetti che hanno tag “Enemy”, “Building” o “Bonus”.

Lo stato di partenza (Avanti) ha due azioni: *Set Float Value* e *Rotate Moduli*. La prima azione imposta il valore della variabile “RotazioneModuli” mentre la seconda azione aggiunge la rotazione al *GameObject* “MODULI_WRAPPER”. Questo stato risponde agli eventi “Stop” e “Bonus”. Nel caso venga lanciato l’evento “Stop” si entrerà nello stato “Indietro” mentre nel caso venga lanciato l’evento “Bonus” la *state machine* entrerà nello stato “Bonus”.

Lo stato “Indietro” si occupa di fermare la rotazione dei moduli e di bloccare l’interazione del personaggio. Infatti al suo interno ha tre azioni: *Animate Float V2*, *Rotate Moduli* e *Send Event to Personaggio*. La prima azione anima la variabile “RotazioneModuli” usando una speciale funzione tramite una curva d’animazione, la seconda azione invece applica la rotazione ai moduli usando la variabile in questione. Andranno spuntate le opzioni *Per Second* ed *Every Frame* per avere una rotazione continua e che sia in base allo scorrere del tempo. La terza azione invece invia l’evento “Disattiva” alla FSM “MoveManager” del personaggio per poter bloccare l’interazione con quest’ultimo. Una volta terminate tutte le azioni l’evento di sistema *FINISHED* attiverà lo stato “Ripartenza”.

Lo stato “Ripartenza” si occupa di gestire e di riavviare la rotazione dei moduli. Le azioni usate sono tre come nello stato precedentemente visto: *Animate Float V2*, *Rotate Moduli* e *Send Event to Personaggio*. Le prime due azioni modificano la variabile di rotazione finchè non tornerà al suo valore originale. L’ultima azione invece invia l’evento “Attiva” al personaggio per reimpostare l’interazione. Una volta terminate tutte le azioni l’evento di sistema *FINISHED* attiverà lo stato iniziale “Avanti”.

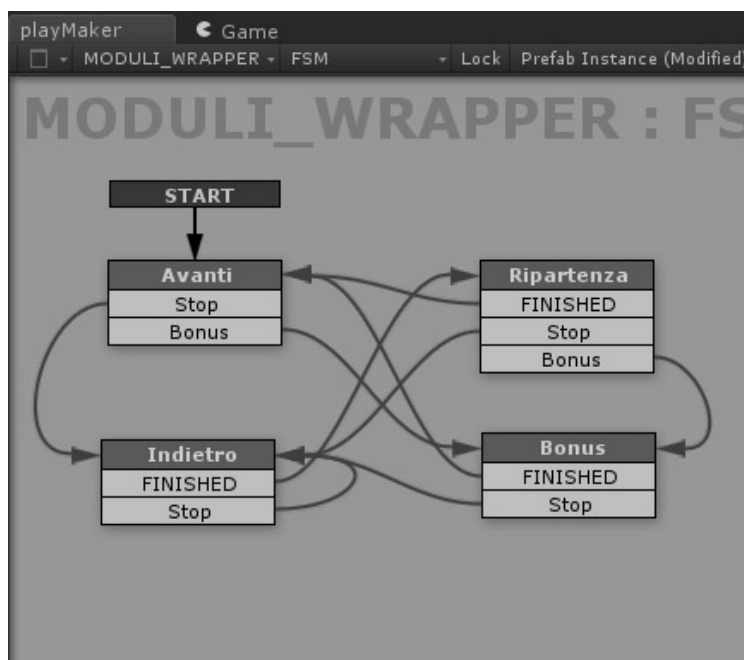


Fig. 5.7 La FSM in MODULI_WRAPPER

5.1.7 Checkpoint

Questa FSM si occupa di distruggere i moduli dopo che il personaggio è passato attraverso il checkpoint relativo.

Nello stato di partenza “listener” non vi è alcuna azione, ma rimane solo in ascolto dell’evento “controllo” che gli verrà inviato dalla FSM presente in “ModuliManager” e più precisamente da uno dei seguenti stati: “Modulo Campagna”, “Modulo Periferia” oppure “Modulo Città”. Una volta lanciato questo evento la *state machine* si posizionerà nello stato “rotazione”.

Nello stato “rotazione” vi sono tre azioni: *Get Parent Checkpoint*, *Get Rotation* e *Float Switch*. La prima azione restituisce il *parent* del checkpoint e lo salva nella variabile “moduloDaDistruggere”. La seconda azione restituisce la rotazione del modulo e la salva nella variabile “rotazioneModulo” mentre la terza azione valuta l’ammontare della rotazione. Se il valore scende sotto il valore di 280 viene lanciato l’evento “distruggi” che attiverà lo stato relativo.

Nello stato “distruggi” vi è una sola azione: *Destroy Object Parent*. Questa azione fa uso della variabile “moduloDaDistruggere” ed eliminerà il GameObject salvato come valore. Una volta terminata l’azione si riattiverà lo stato iniziale “listener”.

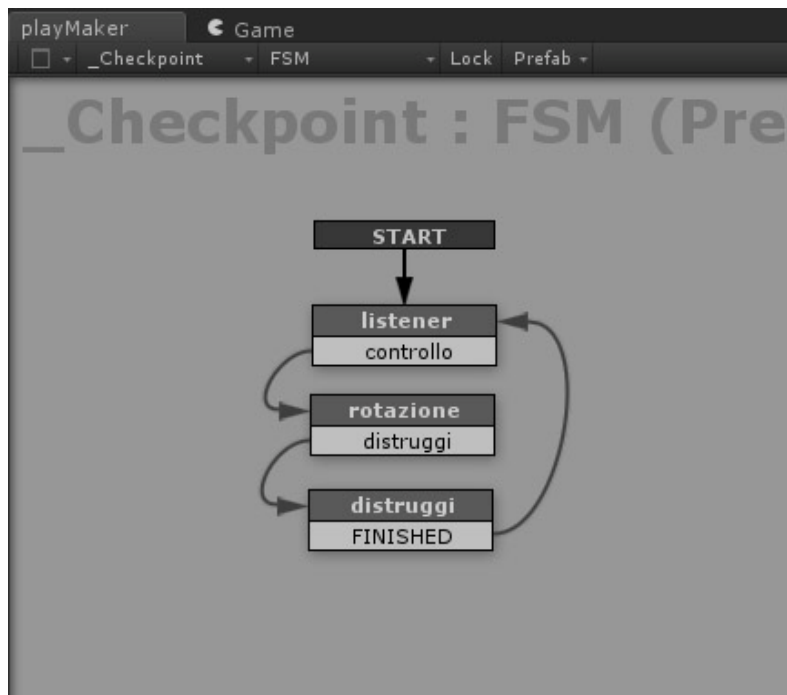


Fig. 5.8 La FSM in Checkpoint

5.1.8 Bonus

La Fsm di questo GameObject si occupa di gestire la collisione con un oggetto che ha come tag “Bonus”.

Lo stato di partenza (Attesa) ha due azioni: *Set Material* e *Set Property*. La prima azione aggiunge il materiale all’oggetto precedentemente impostato nella variabile “MaterialeOggetto”, mentre la seconda azione attiva il *Collider* dell’oggetto per poter lanciare eventi di collisione. Questo stato rimane in attesa dell’evento “Collisione” che una volta avviato posizionerà la FSM nello stato “Colpito”.

Quest'ultimo stato non presenta alcun evento a cui rispondere ma possiede solo un’azione: *Destroy Object*. L’azione si occuperà di distruggere l’oggetto ed eliminarlo dalla scena subito dopo la collisione.



Fig. 5.9 La FSM in Bonus

5.1.9 Nemici – Edifici

Sia negli oggetti con tag “Enemy” che negli oggetti con tag “Building” è presente la stessa *state machine*. Questa FSM si occuperà di gestire il loro comportamento dopo che è avvenuta una collisione.

Nel primo stato (*Attesa*) vi sono due azioni: *Set Material* e *Set Property*. La prima azione aggiunge il materiale impostato nella variabile “MaterialeOggetto” al *GameObject* mentre la seconda azione attiva il *Collider* dell’oggetto. Questo stato rimane in attesa dell’evento “Collisione” che una volta avviato posizionerà la FSM nello stato “Colpito”.

Lo stato “Colpito” rimane in attesa di due eventi: “InAttesa” e “Collisione”. Il primo evento viene attivato automaticamente dopo tre secondi mentre il secondo evento viene attivato nel caso vi siano un’ulteriore collisione. Questo stato inoltre ha tre azioni: *Set Material*, *Set Property* e *Wait*. La prima applica il materiale “TrasparenteRosso” all’oggetto, la seconda lo rende inattivo mentre la terza dopo tre secondi di attesa lancia l’evento specificato.

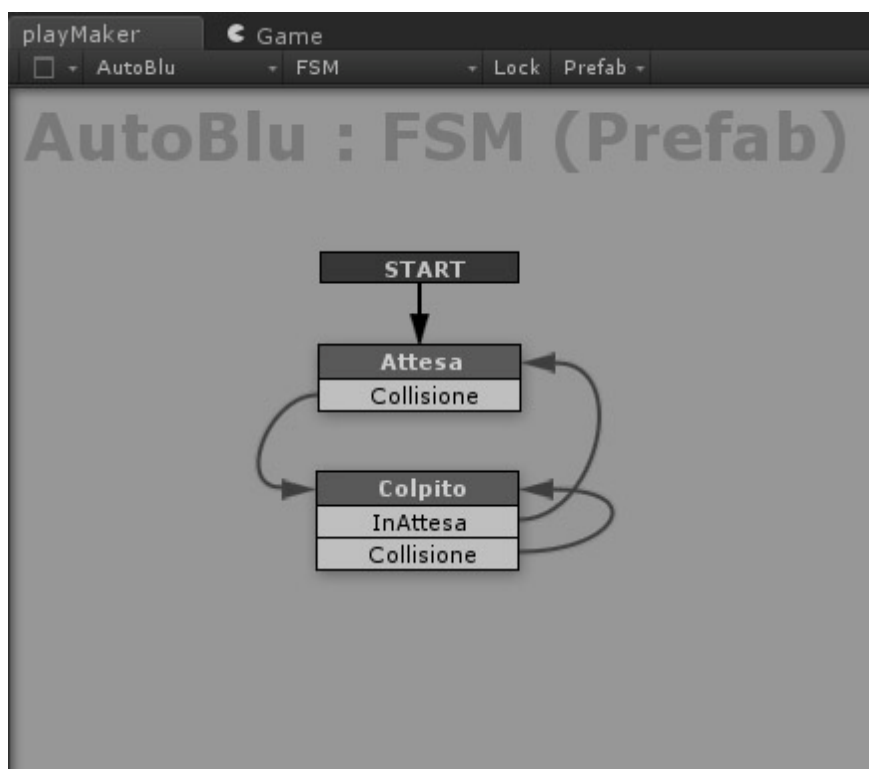


Fig. 3.10 La FSM di AutoBlu

FSM	State
<input checked="" type="checkbox"/> __ColliderManager : FSM_Collider	Listener
<input checked="" type="checkbox"/> __ColliderManager : FSM_Collider (Prefab)	Listener
<input checked="" type="checkbox"/> __GUIManager : Conteggio (Prefab)	Inizializzazione
<input checked="" type="checkbox"/> __GUIManager : Conteggio	Inizializzazione
<input checked="" type="checkbox"/> __ModuliManager : Moduli Partenza	Inizializzazione
<input checked="" type="checkbox"/> __ModuliManager : Moduli Seguenti (Prefab)	Inizializzazione
<input checked="" type="checkbox"/> __ModuliManager : Moduli Seguenti	Listener
<input checked="" type="checkbox"/> __ModuliManager : Partenza (Prefab)	Inizializzazione
<input checked="" type="checkbox"/> __NemiciManager : FSM	Scelta Tipo Nemici
<input checked="" type="checkbox"/> __NemiciManager : FSM (Prefab)	Inizializzazione
<input checked="" type="checkbox"/> _Checkpoint : FSM (Prefab)	listener
<input checked="" type="checkbox"/> _Checkpoint : FSM (Prefab)	listener
<input checked="" type="checkbox"/> _Checkpoint : FSM (Prefab)	listener
<input checked="" type="checkbox"/> _Checkpoint : FSM (Prefab)	listener
<input checked="" type="checkbox"/> _Checkpoint : FSM (Prefab)	listener
<input checked="" type="checkbox"/> AutoBlu : FSM (Prefab)	Attesa
<input checked="" type="checkbox"/> AutoVerde : FSM (Prefab)	Attesa
<input checked="" type="checkbox"/> Bonus : FSM (Prefab)	Attesa
<input checked="" type="checkbox"/> camion : FSM (Prefab)	Attesa
<input checked="" type="checkbox"/> Cielo : FSM (Prefab)	Avanti
<input checked="" type="checkbox"/> Cielo : FSM	Avanti
<input checked="" type="checkbox"/> MODULI_WRAPPER : FSM	Avanti
<input checked="" type="checkbox"/> MODULI_WRAPPER : FSM (Prefab)	Avanti
<input checked="" type="checkbox"/> modulo_C_00_PONTE : FSM (Prefab)	Attesa
<input checked="" type="checkbox"/> modulo_D_02_TRENO : FSM (Prefab)	Attesa
<input checked="" type="checkbox"/> Personaggio : MoveManager	Attivo
<input checked="" type="checkbox"/> Personaggio : TriggerManager	Attivo

Fig. 5.11 Elenco di tutte le FSM

5.2 Script e Algoritmi

5.2.1 Interazione col Personaggio

Le interazioni previste per il nostro personaggio principale sono traslazione sull'asse x e salto verso l'alto lungo l'asse y con possibilità di movimento lungo l'asse x mentre il personaggio è in volo. Questo significa che attraverso l'input da tastiera è possibile muovere il player verso sinistra, verso destra sia quando tocca terra sia quando salta ed è in volo, il tutto nelle coordinate di mondo.

Per poter controllare gli spostamenti del personaggio è stato usato il componente interno di Unity *Character Controller*. Il *Character Controller* viene utilizzato principalmente per un controllo del player in terza persona o in prima persona che non faccia uso della fisica che al contrario viene applicata nel componente di tipo *Rigidbody*. Il *Character Controller*, infatti, non è influenzato da forze e si muoverà solo quando viene chiamata la funzione *Move*. In questo caso eseguirà il movimento che può essere influenzato però da collisioni. Infine per poter applicare delle forze ad oggetti come, ad esempio, spingere in avanti un *Rigidbody* è possibile implementare questa funzione tramite uno script da included dentro la funzione *OnCharacterControllerHit*.

L'uso di questo componente è spiegato dal fatto che i movimenti relativi al player potrebbero essere poco realistici, quindi l'uso di *Rigidbody* (corpi affetti dalle leggi fisiche) potrebbe risultare poco pratico o addirittura non corretto. Infatti, per il nostro personaggio è previsto che possa traslare mentre è in volo dopo che è stata effettuata l'operazione di *Jump* (salto) attraverso la pressione del tasto space. La soluzione è quindi l'utilizzo del *Character Controller* che è composto da un *Collider* (primitive che consentono agli oggetti di entrare in collisione l'uno con l'altro) a forma di capsula che può muoversi in diverse direzioni a seconda dello script creato. Per maggiori informazioni sul componente consultare la pagina: <http://docs.unity3d.com/Documentation/Components/class-CharacterController.html>.

Il codice dello script relativo alla programmazione del controllo del personaggio corrisponde al file denominato "MovingCharacterController.js" e i comportamenti principali sono implementati principalmente dentro la funzione *FixedUpdate*.

Questa particolare funzione di sistema viene chiamata ad intervalli prefissati e permette di aggiornare continuamente la scena di gioco. Infatti, grazie alle sue caratteristiche, consente di avere una scena indipendente dal frame rate.

Proprio per questo è importante menzionare l'utilizzo della proprietà *Time.deltaTime* la quale permette di aggiungere i movimenti del personaggio indipendentemente dal frame rate della scena. Questa variabile contiene l'ammontare di tempo in secondi dall'ultima chiamata alla funzione *FixedUpdate* (tempo in secondi per completare l'ultimo frame).

In questo modo, nel caso ci siano dei rallentamenti durante l'andamento del gioco, i calcoli vengono effettuati in metri al secondo. Questo risultato oltre ad essere significativo è anche più intuitivo poichè gli oggetti si muoveranno con un'unità di misura più simile alla realtà.

Codice sorgente "MovingCharacterController.js":

```
#pragma strict
var speed : float = 70.0;           // velocità di movimento
var jumpSpeed : float = 30.0;      // velocità di salto
var gravity : float = 35.0;        // forza di gravità
var traslazione : float = 0;        // forza applicata se collide
var assey : int = 10;              // forza applicata se collide
var moveDirection = Vector3.zero;  // inizializzo il movimento
var grounded : boolean = false;    // atterraggio
var salto : boolean = true;        // attiva / disattiva salto

function Update () {
    transform.position.z = -42.5;
}

function FixedUpdate () {
    moveDirection.x = Input.GetAxis("Horizontal") * speed;
    if (grounded) {
        moveDirection = new Vector3(Input.GetAxis("Horizontal"),0,0);
        moveDirection = transform.TransformDirection(moveDirection);
        moveDirection.x *= speed;
        moveDirection.x -= traslazione;
        if (Input.GetButton ("Jump") && salto) {
            moveDirection.y = jumpSpeed;
        }
    }
    var c:CharacterController = GetComponent(CharacterController);
    moveDirection.y -= assey;
    moveDirection.y -= gravity * Time.deltaTime;
    var flags = c.Move(moveDirection * Time.deltaTime);
    grounded = (flags & CollisionFlags.CollidedBelow) != 0;
}
@script RequireComponent(CharacterController)
```

5.2.2 Randomizzazione degli Ostacoli

Questo script si occupa di posizionare gli oggetti nemici sui piani a disposizione del modulo caricato. Gli oggetti nemici vengono presi a random da un'array di *GameObject* e poi vengono posizionati sui piani disponibili allineando l'asse z alla normale al piano. Prima di tutto, per poter istanziare gli ostacoli sui moduli di gioco, è stato necessario posizionare dei piani in posizioni prefissate sui moduli e successivamente creare uno script che si occupasse di creare gli oggetti e di posizionarli a random sui piani in questione.

Lo script in questione utilizza tre funzioni di sistema: *Awake*, *Start*, e *Update*.

La funzione *Awake* viene chiamata per prima e si occupa di inizializzare le variabili “myObject” dello script. Viene creato un riferimento alla state machine “NemiciManager” la quale restituirà il valore della variabile “nemico” riferita alla scena di gioco attuale che verrà salvata in “myObject”.

Nella funzione *Start* viene inizializzato il tempo necessario ad istanziare gli oggetti e viene creato l'array di *GameObject* denominato “spawnPoint” grazie alla funzione *FindGameObjectsWithTag("...")*. Questa funzione restituirà un array di *GameObject* che hanno come tag “piano”. Vengono inoltre inizializzate le variabili “objectCount”, “pianiRimanti” e “modulo”. È importante menzionare che modulo si riferisce all'oggetto che contiene lo script.

Il funzionamento principale dello script avviene nella funzione *Update* che verrà chiamata continuamente ad ogni frame di gioco. In questa funzione vengono istanziati gli oggetti nemici restituiti dalla state machine inizializzata precedentemente finché il numero degli oggetti istanziati non è minore del valore presente nella variabile “numberToInstantiate”. Inoltre nella funzione *Update* avviene il processo di posizionamento a random su uno dei piani del modulo. La funzione si occuperà di allineare gli oggetti creati alla normale alla superficie dei piani attraverso l'uso del *Raycasting*.

Sempre all'interno della funzione *Update*, vi è controllo relativo ai piani rimanenti ai quali deve essere modificato il tag relativo da “Piano” ad “Untagged” in modo che lo stesso script presente in un altro modulo non vada in conflitto con i piani

presente in quello precedente. Infine una volta che sono stati posizionati tutti gli oggetti lo script viene disabilitato.

Codice sorgente "RandomObject.js":

```
#pragma strict

static var numeroMinOggetti: int = 1; // numero min oggetti
static var numeroMaxOggetti: int = 3; // numero Max oggetti
static var numberToInstantiate : int; // numero Max oggetti
var numeroNemici : int; // numero nemici
var myObject : GameObject; // oggetti da istanziare
var spawnPoint : GameObject[]; // creazione array
var randomizzazione :int; // variabile casuale
var pianiRimanenti: int; // piani non usati
var nemiciManager : PlayMakerFSM; // fsm nemiciManager
private var minTime : float = 0.001; // tempo minimo
private var MaxTime : float = 0.005; // tempo Max
var modulo : Transform; // modulo
private var waitTime : float; // tempo calcolato
private var objectCount : int; // oggetti istanziati

function Awake() {
    nemiciManager=GameObject.Find("__NemiciManager").
    GetComponent.<PlayMakerFSM>();
    myObject=nemiciManager.FsmVariables.
    GetFsmGameObject("nemico").Value;
}

function Start () {
    waitTime = Random.Range (minTime, MaxTime);
    objectCount = 0;
    spawnPoint = GameObject.FindGameObjectsWithTag("Piano");
    pianiRimanenti = spawnPoint.Length - numberToInstantiate;
    modulo = this.gameObject.transform;
}

function Update () {
    numeroNemici = numberToInstantiate;
    if (Time.time>waitTime && objectCount<numberToInstantiate) {

        waitTime = Time.time+Random.Range(minTime, MaxTime);
        var obj : GameObject;

        nemiciManager = GameObject.Find("__NemiciManager").
        GetComponent.<PlayMakerFSM>();

        nemiciManager.Fsm.Event("ricarica");

        myObject = nemiciManager.FsmVariables.
        GetFsmGameObject("nemico").Value;

        randomizzazione = Random.Range (0, spawnPoint.Length);

        if (spawnPoint[randomizzazione].tag == "Piano") {
            obj = Instantiate(myObject,
            spawnPoint[randomizzazione].transform.position,
```

```

spawnPoint[randomizzazione].transform.rotation) as
GameObject;

obj.transform.parent = modulo;
var hit : RaycastHit;

// RAYCASTING
if(Physics.Raycast(obj.transform.position,-Vector3.up,
hit)) {
    // WORLD SPACE
    obj.transform.rotation = Quaternion.
    FromToRotation(Vector3.forward, hit.normal);

    // SELF SPACE
    obj.transform.Rotate(0, 0, 90);
    Random.Range (-8, 8);
    obj.transform.Translate(0, spostamento, -0.8);

    if (obj.tag == "Bonus") {
        obj.transform.Translate(0, 0, 5);
    }

    spawnPoint[randomizzazione].tag = "Untagged";
    spawnPoint[randomizzazione].renderer.enabled =
false;

    objectCount ++;
}
else {
    Destroy(obj.transform.transform.gameObject);
}
}
spawnPoint = GameObject.FindGameObjectsWithTag("Piano");
}

if (spawnPoint.Length == pianiRimanenti) {
    for(var i:int=0;i<spawnPoint.Length;i++) {
        if (spawnPoint[i].tag == "Piano") {
            spawnPoint[i].tag = "Untagged";
            spawnPoint[i].renderer.enabled = false;
        }
    }
}

if (GameObject.FindGameObjectsWithTag("Piano").Length == 0) {
    var script = this.GetComponent(_RandomObject);
    script.enabled = false;
}
}

@script AddComponentMenu ("MOVEIT/Random Object")

```


5.2.3 Start

Questo script si occupa di creare un conto alla rovescia iniziale (mostrato a video attraverso l'interfaccia) prima che il videogioco parta e che l'utente possa iniziare effettivamente a giocare. Lo script in questione è stato implementato per avere il tempo di prepararsi a giocare e per avere una visuale effettiva della scena.

Vengono utilizzate due funzioni di sistema, *Awake* e *OnGUI*, più altre due funzioni che comunicano con altri script, "Disattiva" e "Attiva".

Nelle prima funzione vengono inizializzati il tempo, la scalatura del tempo, una variabile relativa al nome del livello ed altre due variabili booleane.

Nella seconda funzione invece avviene il conto alla rovescia che una volta finito avvierà effettivamente il gioco permettendo all'utente di interagire.

Le ultime due funzioni invece si occupano di gestire gli script "Pausa.js" e "PausaVite.js" in modo che non vadano in conflitto con il conto alla rovescia iniziale. Una volta finito il conteggio viene avviato uno o l'altro script a seconda della modalità di gioco che abbiamo scelto, a "tempo" o a "vite".

Codice sorgente "Start.js":

```
#pragma strict

private var startTime : float;           // tempo partenza
private var restSeconds : int;           // secondi rimasti
private var roundedRestSeconds : int;    // arrotondamento
private var displaySeconds : int;        // secondi nel display
private var guiMessage : GameObject;     // gui hurry up
private var guiTime : float;
private var attiva : boolean;
private var disattiva : boolean;
var tipoGioco : String;
var countdownSeconds : int;              // tempo iniziale
var guiManager : PlayMakerFSM;          // fsm guimanager
var globalVariables;                     // variabili globali
var numeroCheckPoint : int;              // checkpoint passati
var difficoltaCheckPoint : int;          // checkpoint da passare

function Awake () {
    disattiva = false;
    attiva = false;
    Time.timeScale = 0.0000001;
    startTime = Time.timeSinceLevelLoad;
    tipoGioco = Application.loadedLevelName;
}
}
```

```

function OnGUI () {
    Disattiva();
    var partenza : boolean = false;
    var text : String = displaySeconds.ToString();
    guiMessage = GameObject.Find("GUI_Partenza");
    guiMessage.guiText.material.color = Color.red;
    guiMessage.guiText.fontSize = 65;
    if (!partenza) {
        guiTime = (Time.timeSinceLevelLoad*10000000) -
            startTime;
        restSeconds = 4 - (guiTime);
        guiMessage.GetComponent(GUIText).text = text;
    }
    if (restSeconds > 0) {
        guiTime = (Time.timeSinceLevelLoad*10000000) -
            startTime;
        restSeconds = 4 - (guiTime);
        guiMessage.GetComponent(GUIText).text = text;
        partenza = true;
    }
    if (restSeconds <= 0) {
        guiManager = GameObject.Find("__GUIManager").
            GetComponent.<PlayMakerFSM>();
        numeroCheckPoint =
            guiManager.FsmVariables.GlobalVariables.
            GetFsmInt("CheckPointPassed").Value;
        guiMessage.guiText.fontSize = 45;
        guiMessage.guiText.text = "PARTENZA !!";
        Time.timeScale = 1;
        guiTime = Time.timeSinceLevelLoad;
        restSeconds = countdownSeconds - (guiTime);
    }
    if (restSeconds <= -1) {
        guiMessage.guiText.text = "";
        guiTime = Time.timeSinceLevelLoad;
        restSeconds = countdownSeconds - (guiTime);
        this.enabled = false;
        Attiva();
    }
    roundedRestSeconds = Mathf.CeilToInt(restSeconds);
    displaySeconds = roundedRestSeconds % 60;
}
function Disattiva() {
    if (!disattiva) {
        this.GetComponent(_Pausa).enabled = false;
        this.GetComponent(_PausaVite).enabled = false;
        disattiva = true;
    }
}
function Attiva() {
    if (!attiva) {
        if (tipoGioco == "tempo") {
            this.GetComponent(_Pausa).enabled = true;
        }
        else if (tipoGioco == "vite") {
            this.GetComponent(_PausaVite).enabled = true;
        }
    }
}
}

```

5.2.4 Count Down

Nella modalità di gioco a “tempo” è stato necessario implementare uno script che tenesse conto dello scorrere del tempo per poter creare i controlli relativi che permettono di gestire la fine di un livello, la difficoltà e il numero dei nemici presenti sulla scena.

In questo script sono state usate tre funzioni di sistema, *Awake*, *Update* e *OnGui*, per gestire i controlli ed un'altra funzione, “Proporzioni”, per creare la barra di avanzamento della scena.

Nella prima funzione (*Awake*) vengono inizializzati il tempo, il *targetFrame*, una variabile booleana (*fineLivello*) e altre due variabili relative al caricamento delle immagini necessarie alla barra di progresso.

Nella seconda funzione (*Update*) vengono impostate le proporzioni della barra attraverso la chiamata alla funzione Proporzioni che eseguirà i calcoli necessari. Inoltre viene continuamente aggiornato il valore della variabile “numeroCheckPoint” creando un riferimento alla state machine interna al *GameObject* “GUI Manager” la quale passerà il valore della variabile globale “CheckPointPassed”.

Infine nella funzione *OnGUI* sono stati implementati: i controlli relativi allo scorrere del tempo, l'interfaccia grafica della barra di avanzamento e l'interfaccia grafica del campo di testo che visualizzerà i secondi rimasti. Uno dei controlli più importanti della funzione *OnGUI* è quello relativo al numero di checkpoint: se il valore della variabile “numeroCheckPoint” coincide col valore della variabile “difficoltaCheckPoint” verrà attivata una pausa ed una chiamata allo script “Pausa.js” per rendere possibile una visualizzazione del punteggio parziale.

Codice sorgente di “CountDown.js”:

```
#pragma strict
private var startTime : float;           // tempo
private var restSeconds : int;           // secondi rimanenti
private var displaySeconds : int;        // secondi DISPLAY
private var displayMinutes : int;        // minuti DISPLAY
private var fineLivello : boolean;       // controllo
static var roundedRestSeconds : int;     // variabile globale
// -----
```

```

private var guiCount : GameObject;           // gui countdown
private var guiHurryUp : GameObject;        // gui hurry up
var countDownSeconds : int = 61;           // tempo iniziale
var guiManager : PlayMakerFSM;             // fsm guimanager
var globalVariables;                       // variabili globali
// -----
static var difficoltaCheckPoint : int;     // checkpoint da passare
static var numeroCheckPoint : int;        // checkpoint passati
// -----
var progress : float = 0;                  // barra progresso
var pos : Vector2 = new Vector2();         // posizione barra
var size : Vector2 = new Vector2();        // dimensione barra
var progressBarEmpty : Texture2D;         // texture rossa
var progressBarFull : Texture2D;          // texture verde
private var larghezzaSchermo : float;     // Screen.width
private var altezzaSchermo : float;       // Screen.height

function Awake () {
    Application.targetFrameRate = 60;
    startTime = Time.time;
    fineLivello = false;
    progressBarEmpty = Resources.Load("progress_bar_red");
    progressBarFull = Resources.Load("progress_bar_green");
}

function Proporzioni (proporzione: float, asse: String) {
    var calcolo : float;

    if (asse == "x") {
        larghezzaSchermo = Screen.width;
        calcolo = larghezzaSchermo * proporzione;
    }
    else if (asse == "y") {
        altezzaSchermo = Screen.height;
        calcolo = altezzaSchermo * proporzione;
    }
    return calcolo;
}

function Update () {

    pos.x = Proporzioni(0.03, "x");
    pos.x += 0;
    pos.y = Proporzioni(0.03, "y");
    pos.y += 0;
    size.x = Proporzioni(0.93, "x");
    size.y = 2;

    guiManager = GameObject.Find("__GUIManager").
    GetComponent.<PlayMakerFSM>();

    numeroCheckPoint = guiManager.FsmVariables.GlobalVariables.
    GetFsmInt("CheckPointPassed").Value;

    progress = numeroCheckPoint * (size.x/difficoltaCheckPoint);
}

```

```

function OnGUI () {

    GUI.DrawTexture(Rect(pos.x, pos.y, size.x, size.y),
    progressBarEmpty);

    GUI.BeginGroup(new Rect (pos.x, pos.y, progress, size.y));

    GUI.DrawTexture(new Rect(0, 0, size.x, size.y),
    progressBarFull);

    GUI.EndGroup();

    guiCount = GameObject.Find("GUI_CountDown");
    guiHurryUp = GameObject.Find("GUI_Sbrigati");

    var text = String.Format ("{0:00}:{1:00}", displayMinutes,
    displaySeconds);

    var guiTime = Time.time - startTime;
    restSeconds = countdownSeconds - (guiTime);

    if (restSeconds == 60) {
        guiCount.guiText.material.color = Color.white;
    }
    if (restSeconds == 58) {
        guiHurryUp.guiText.text = "";
    }
    if (restSeconds == 20) {
        guiHurryUp.guiText.text = "SBRIGATI !!";
        guiHurryUp.guiText.material.color = Color.red;
        guiCount.guiText.material.color = Color.red;
    }
    if (restSeconds == 18) {
        guiHurryUp.guiText.text = " ";
    }
    // -----
    // VITTORIA
    // -----
    if (numeroCheckPoint==difficoltaCheckPoint && !fineLivello){
        fineLivello = true;
        var script = this.GetComponent(_CountDown);
        // FINE LIVELLO
        if (_Pausa.scenaAttuale<(_Pausa.elencoScene.length-1)){
            guiHurryUp.guiText.transform.position.y = 0.9;
            guiHurryUp.guiText.material.color = Color.green;
            guiHurryUp.guiText.fontSize = 30;
            guiHurryUp.guiText.text = "BRAVO !! PROSEGUI ..";
            _Pausa.isWinner = true;
            _Pausa.scenaAttuale += 1;
            script.enabled = false;
        }
        // FINE GIOCO
        else {
            guiHurryUp.guiText.transform.position.y = 0.9;
            guiHurryUp.guiText.material.color = Color.green;
            guiHurryUp.guiText.fontSize = 30;
            guiHurryUp.guiText.text = "HAI VINTO !!";
            _Pausa.isFinished = true;
            _Pausa.scenaAttuale += 1;
        }
    }
}

```

```

        script.enabled = false;
    }

}

// SCONFITTA
if (restSeconds == 0) {

    if (numeroCheckPoint < difficoltaCheckPoint) {
        guiManager = GameObject.Find("__GUIManager").
            GetComponent.<PlayMakerFSM>();

        numeroCheckPoint =
            guiManager.FsmVariables.GlobalVariables.
            GetFsmInt("CheckPointPassed").Value;

        guiHurryUp.guiText.transform.position.y = 0.8;
        guiHurryUp.guiText.material.color = Color.red;
        guiHurryUp.guiText.fontSize = 30;
        guiHurryUp.guiText.text = "HAI PERSO !!";
        _Pausa.isLooser = true;
        this.enabled = false;
    }

}

// visualizza minuti e secondi nel display
roundedRestSeconds = Mathf.CeilToInt(restSeconds);
displaySeconds = roundedRestSeconds % 60;
displayMinutes = roundedRestSeconds / 60;
guiCount.GetComponent(GUIText).text = text;
}

```

5.2.5 Pausa

La funzionalità di questo script è quella di creare una pausa durante lo svolgimento del gioco con due diverse modalità: la prima avviene attraverso la pressione del pulsante “Esc” da parte dell’utente, l’altra in automatico ogni volta che si finisce un livello intermedio.

Per implementare questo script è stato fatto uso delle funzioni di sistema *Awake*, *Update* e *OnGUI*. Inoltre sono state implementate altre due funzioni: “CalcolaPunteggiParziali” e “ResetScena”.

Nella funzione *Awake* vengono inizializzati la scena e la state machine “moduliManager” la quale restituisce la variabile globale “moduliDaCaricareGlobale” per poter salvare il livello iniziale.

Nella funzione *Update* viene controllata la pressione del pulsante “Esc” attraverso il quale è possibile attivare / disattivare la pausa. Inoltre viene eseguito nu

controllo sulle variabili globali “isLooser”, “isWinner” e “isFinished” tramite la quali il gioco viene messo in pausa se il giocatore ha perso, se ha passato un livello intermedio oppure se ha finito un livello. Le pause intermedie vengono usate per evidenziare i punteggi parziali tra un livello e l’altro.

Nella funzione *OnGUI* è stata implementata tutta la gestione dell’interfaccia grafica. Vengono create e gestite le seguenti interfacce grafiche: “pausa”, “sconfitta”, “vittoria” e “gioco finito”. Per ognuna di queste interfacce saranno visualizzati a video pulsanti e testi differenti a seconda della situazione in cui l’utente si troverà.

Le ultime due funzioni servono una a calcolare i punteggi parziali nel passaggio da un livello all’altro e l’altra a resettare la scena nel caso si voglia incominciare da capo il gioco.

Codice sorgente di Pausa.js:

```
#pragma strict

var guiSkin: GUISkin; // skin pulsanti
private var guiMessage : GameObject; // gui
private var guiHurryUp : GameObject; // gui
private var guiCount : GameObject; // gui
private var guiScore : GameObject; // gui
private var guiBonus : GameObject; // gui

// GESTIONE DEI MODULI
private var guiManager : PlayMakerFSM; // fsm gui
private var moduliManager : PlayMakerFSM; // fsm moduli
private var nemiciManager : PlayMakerFSM; // fsm nemici
var moduliStart : String; // modulo partenza
private var globalVariables; // var globale
static var scenaAttuale : int; // scena attuale
static var elencoScene : Array = ["Campagna","Periferia","Città"];

// GESTIONE PAUSE
static var isPaused : boolean = false; // var globale
static var isLooser : boolean = false; // var globale
static var isWinner : boolean = false; // var globale
static var isFinished : boolean = false; // var globale

// PUNTEGGI
var seconds : int;
var check : int;
var valoreBonus: int;
var plus : int;
var bonus : int;
var totale : int;
```

```

function Awake () {
    scenaAttuale = 0;
    moduliManager = GameObject.Find("__ModuliManager").
    GetComponent.<PlayMakerFSM>();
    moduliStart = moduliManager.FsmVariables.GlobalVariables.
    GetFsmString("moduliDaCaricareGlobale").Value;
}
function Update() {

    if(Input.GetKeyDown("escape") && !isPaused && !isLooser &&
    !isWinner && !isFinished) {
        print("Paused");
        Time.timeScale = 0.0;
        isPaused = true;
    }
    else if(Input.GetKeyDown("escape") && isPaused && !isLooser &&
    !isWinner && !isFinished) {
        print("Unpaused");
        Time.timeScale = 1.0;
        isPaused = false;
        GameObject.Find("Camera").
        GetComponent(DepthOfFieldScatter).aperture = 7;
    }
    else if(isLooser) {
        print("Sconfitta");
        Time.timeScale = 0.0;
    }
    else if(isWinner) {
        print("Vittoria");
        Time.timeScale = 0.005;
    }
    else if(isFinished) {
        print("Fine!");
        Time.timeScale = 0.0;
    }
}

function OnGUI () {
    var wButton : int = 160; // larghezza pulsanti

    guiHurryUp = GameObject.Find("GUI_Sbrigati");
    guiMessage = GameObject.Find("GUI_Message");
    guiMessage.guiText.text = "";
    GUI.skin = guiSkin;

    if(isPaused) {
        GameObject.Find("Camera").
        GetComponent(DepthOfFieldScatter).
        aperture = 100;
        guiMessage.guiText.transform.position.y = 0.85;
        guiMessage.guiText.material.color = Color.white;
        guiMessage.guiText.fontSize = 35;
        guiMessage.guiText.text = "Pausa";

        if(GUI.Button (Rect((Screen.width-wButton)/2, (Screen.height-
        30)/2-60, wButton, 30), "Continua a giocare", "button")) {
            print("Continue");
            Time.timeScale = 1.0;
        }
    }
}

```



```

        isPaused = false;
        GameObject.Find("Camera").
        GetComponent (DepthOfFieldScatter).
        aperture = 7;
    }
    if(GUI.Button (Rect((Screen.width-wButton)/2,(Screen.height-
30)/2-20,wButton,30), "Reinizia il Livello", "button")) {
        print("Restart");
        // RESET SCENA / MODULI
        ResetScena();
        Application.LoadLevel("tempo");
        Time.timeScale = 1.0;
        isPaused = false;
        GameObject.Find("Camera").
        GetComponent (DepthOfFieldScatter).
        aperture = 7;
    }
    if(GUI.Button (Rect((Screen.width-wButton)/2,(Screen.height-
30)/2+20,wButton,30), "Torna al Menu Principale", "button")) {
        print("Menu");
        // RESET SCENA / MODULI
        ResetScena();
        Time.timeScale = 1.0;
        isPaused = false;
        Application.LoadLevel("start");
    }
    if(GUI.Button (Rect((Screen.width-wButton)/2,(Screen.height-
30)/2+60,wButton,30), "Esci dal Gioco")) {
        print("Quit!");
        Application.Quit();
    }
}

if(isLooser) {

    GameObject.Find("Camera").
    GetComponent (DepthOfFieldScatter).
    aperture = 100;

    if(GUI.Button (Rect((Screen.width-wButton)/2,(Screen.height-
30)/2-40,wButton,30), "Reinizia il Livello", "button")) {
        print("Restart");
        // RESET SCENA / MODULI
        ResetScena();
        isLooser = false;
        Time.timeScale = 1.0;
        GameObject.Find("Camera").
        GetComponent (DepthOfFieldScatter).
        aperture = 7;
        Application.LoadLevel("tempo");
    }
    if(GUI.Button (Rect((Screen.width-wButton)/2,(Screen.height-
30)/2,wButton,30), "Torna al Menu Principale", "button")) {
        print("Menu");
        // RESET SCENA / MODULI
        ResetScena();
        isLooser = false;
        Time.timeScale = 1.0;
        Application.LoadLevel("start");
    }
}

```



```

// 5) REINIZIALIZZO CHECKPOINT E BONUS COLPITI
guiManager = GameObject.Find("__GUIManager").
GetComponent.<PlayMakerFSM>();
guiManager.FsmVariables.GlobalVariables.
GetFsmInt("CheckPointPassed").Value = 0;

guiManager.FsmVariables.
GetFsmInt("bonusColpiti").Value = 0;

// 6) TOGLIERE BLUR
GameObject.Find("Camera").
GetComponent(DepthOfFieldScatter).
aperture = 7;

// 7) CAMBIARE SCRITTA
guiHurryUp.guiText.transform.position.y = 0.5;
guiHurryUp.guiText.material.color = Color.green;
guiHurryUp.guiText.fontSize = 40;
guiHurryUp.guiText.text = "Ora sei in\n" +
elencoScene[scenaAttuale];
}
GUI.EndGroup ();
}

if(isFinished) {

    CalcolaPunteggiParziali();
    // SALVO PUNTEGGIO
    _Punteggio.arrayPunteggiScenari[scenaAttuale-1] =
totale;
    GameObject.Find("Camera").
    GetComponent(DepthOfFieldScatter).aperture = 100;

    GUI.BeginGroup(Rect(Screen.width / 2 - 125,
Screen.height / 2 - 100, 250, 200));
    GUI.Box(Rect(0, 0, 250, 200), "Punteggio Finale");
    GUI.Label (Rect(20, 30, 200, 30), elencoScene[0] +
": " + _Punteggio.arrayPunteggiScenari[0] );
    GUI.Label (Rect(20, 50, 200, 30), elencoScene[1] +
": " + _Punteggio.arrayPunteggiScenari[1] );
    GUI.Label (Rect(20, 70, 200, 30), elencoScene[2] +
": " + _Punteggio.arrayPunteggiScenari[2] );
    GUI.Label (Rect(20, 90,wButton,30), "Punteggio: " +
GetComponent(_Punteggio).PunteggioFinale());

    if(GUI.Button (Rect(25,120,200,30), "Gioca di
nuovo!!", "button")) {
        print("Restart!");
        isFinished = false;
        Time.timeScale = 1.0;
        GameObject.Find("Camera").
        GetComponent(DepthOfFieldScatter).aperture = 7;
        // RESET SCENA / MODULI
        ResetScena();
        Application.LoadLevel("tempo");
    }
    if(GUI.Button (Rect(25,160,200,30), "Torna al Menu

```

```

        Principale", "button")) {
        print("Menu");
        isFinished = false;
        Time.timeScale = 1.0;

        // RESET SCENA / MODULI
        ResetScena();
        Application.LoadLevel("start");
    }
    GUI.EndGroup ();
}
}

function CalcolaPunteggiParziali() {
    // CALCOLO PUNTEGGIO:
    seconds = _CountDown.roundedRestSeconds+1;
    check = _CountDown.numeroCheckPoint;
    guiManager = GameObject.Find("__GUIManager").
    GetComponent.<PlayMakerFSM>();
    valoreBonus = guiManager.FsmVariables.
    GetFsmInt("bonusColpiti").Value;
    plus = (seconds*100/60);
    bonus = valoreBonus*10;
    totale = (seconds*check) + plus + bonus;
}

function ResetScena() {
    scenaAttuale = 0;
    moduliManager.FsmVariables.GlobalVariables.
    GetFsmString("moduliDaCaricareGlobale").Value = moduliStart;
}

@script AddComponentMenu ("GUI/Pause GUI")

```

5.2.6 Vite

Questo script è stato creato per la gestione della modalità di gioco a “vite”. Durante questa modalità il giocatore deve fare il possibile per non scontrarsi con gli oggetti sul percorso, pena la perdita di una delle tre vite a disposizione.

Per l'implementazione sono state usate le tre funzioni di sistema *Awake*, *Update* e *OnGUI*, più altre tre funzioni “PrimaCollisione”, “SecondaCollisione” e “Pulisci”.

Nella funzione *Awake* vengono inizializzati il tempo e le vite inoltre vengono referenziati oggetti della GUI e due delle *state machine* che saranno utilizzate rispettivamente per interagire con la GUI Manager e col Personaggio.

Nella funzione *Update* viene semplicemente aggiornato il valore della variabile “numeroVite” passando il valore della variabile “viteAttuali” presente nella *state machine* interna alla GUI Manager.

Nella funzione *OnGUI* vengono gestite le diverse casistiche di gioco in cui l'utente avrà a disposizione rispettivamente tre vite, due vite, una vita e zero vite. Nell'ultimo caso il gioco finisce e viene impostata la variabile globale "isFinished" a true in modo che si attivi una pausa e il giocatore possa vedere il punteggio relativo.

Le ultime tre funzioni (PrimaCollisione – SecondaCollisione – Pulisci) servono a gestire l'interazione col personaggio che dopo ogni collisione verrà disattivato per un tempo di tre secondi durante il quale verrà mostrato un messaggio a schermo, verrà disattivato il movimento e sarà resa invisibile la *mesh* del personaggio. Finito questo conteggio si potrà interagire di nuovo e continuare a giocare.

Codice sorgente di Vite.js:

```
#pragma strict

private var startTime : float; // inizializzo tempo
static var roundedSeconds : int; // var globale secondi
static var scoreSeconds : int; // var globale secondi
var tempoColpito : int = 0; // var tempo
var pulisci : boolean = false; // var booleana
var prima : boolean = false; // var booleana
var seconda : boolean = false; // var booleana
var customGuiStyle : GUIStyle; // var GUIStyle
static var numeroVite : int = 3; // numero vite
var guiManager : PlayMakerFSM; // PlayMakerFSM
var personaggio : PlayMakerFSM; // PlayMakerFSM
private var guiCount : GameObject; // gui countdown
private var guiHurryUp : GameObject; // gui hurry up

function Awake () {
    _PausaVite.isFinished = false;
    numeroVite = 3;
    startTime = Time.time;
    guiCount = GameObject.Find("GUI_CountDown");
    guiHurryUp = GameObject.Find("GUI_Sbrigati");

    // FSM
    guiManager = GameObject.Find("__GUIManager").
    GetComponent.<PlayMakerFSM>();
    personaggio = GameObject.Find("PersonaggioVite").
    GetComponent.<PlayMakerFSM>();
    numeroVite = guiManager.FsmVariables.
    GetFsmInt("viteAttuali").Value;
}

function Update () {
    numeroVite = guiManager.FsmVariables.
    GetFsmInt("viteAttuali").Value;
}
```

```

function OnGUI () {
    Pulisci();
    var text = String.Format ("{0:0000}", scoreSeconds);
    var guiTime = Time.time - startTime;
    roundedSeconds = Mathf.CeilToInt(guiTime);
    scoreSeconds = guiTime * 10;
    guiCount.GetComponent(GUIText).text = text;
    switch(numeroVite){
        case 3:
            GUI.Label (Rect(Screen.width - 140, 20, 40, 52),
            "", customGuiStyle);
            GUI.Label (Rect(Screen.width - 100, 20, 40, 52),
            "", customGuiStyle);
            GUI.Label (Rect(Screen.width - 60, 20, 40, 52),
            "", customGuiStyle);
            break;

        case 2:
            PrimaCollisione();
            GUI.Label (Rect(Screen.width - 100, 20, 40, 52),
            "", customGuiStyle);
            GUI.Label (Rect(Screen.width - 60, 20, 40, 52),
            "", customGuiStyle);
            break;

        case 1:
            SecondaCollisione();
            GUI.Label (Rect(Screen.width - 60, 20, 40, 52),
            "", customGuiStyle);
            break;

        case 0:
            guiHurryUp.guiText.transform.position.y = 0.9;
            guiHurryUp.guiText.fontSize = 30;
            guiHurryUp.guiText.text = "FINE GIOCO! RITENTA";
            _PausaVite.isFinished = true;
            var script = this.GetComponent(_Vite);
            script.enabled = false;
            break;
    }
}

function PrimaCollisione() {
    if(!prima){
        tempoColpito = roundedSeconds;
        prima = true;
        pulisci = true;
    }
}

function SecondaCollisione() {
    if(!seconda){
        tempoColpito = roundedSeconds;
        seconda = true;
        pulisci = true;
    }
}

function Pulisci() {
    if (pulisci) {
        guiHurryUp.guiText.text = "COLPITO !!";
        if ((roundedSeconds-tempoColpito) >= 3) {

```

```

        guiHurryUp.guiText.text = " ";
        personaggio.Fsm.Event("AttivaPlayer");
        pulisci = false;
    }
}
}

```

5.2.7 PausaVite

La funzionalità di questo script è quella di creare una pausa durante lo svolgimento del gioco a “vite” in due diversi modi: il primo modo avviene attraverso la pressione del pulsante “Esc” da parte dell’utente, l’altro in automatico ogni volta che il gioco finisce.

Per l’implementazione è stato fatto uso delle funzioni di sistema *Awake*, *Update* e *OnGUI*. Inoltre sono state implementate altre due funzioni: “CalcolaPunteggiFinali” e “ResetScena”.

Nella funzione *Awake* vengono inizializzati la scena, la state machine presente in “moduliManager” (restituisce la variabile globale “moduliDaCaricareGlobale” per poter salvare il livello iniziale) e la state machine presente in “guiManager” (restituisce un riferimento alla variabile “vitaAttuali” per ottenere il numero di vite disponibili).

Nella funzione *Update* viene controllata la pressione del pulsante “Esc” attraverso la quale è possibile attivare / disattivare la pausa. Inoltre viene eseguito un controllo sulla variabile globale “isFinished”. Se il giocatore ha perso tutte e tre le vite di gioco il valore viene impostato a “true” e il gioco viene messo in pausa.

Nella funzione *OnGUI* è stata implementata tutta la gestione dell’interfaccia grafica. Vengono create e gestite le seguenti interfacce grafiche: “pausa”, e “gioco finito”. Per ognuna di queste interfacce saranno visualizzati a video pulsanti e testi differenti a seconda della situazione in cui l’utente si troverà.

Le ultime due funzioni servono rispettivamente per gestire il calcolo del punteggio finale e per resettare la scena una volta che il gioco ricomincia da capo.

Codice sorgente di PausaVite.js:

```

#pragma strict
var guiSkin: GUISkin; // skin pulsanti

```

```

private var guiMessage : GameObject;           // gui
private var guiHurryUp : GameObject;          // gui
private var guiCount : GameObject;            // gui
private var guiScore : GameObject;            // gui
private var guiBonus : GameObject;            // gui

// GESTIONE DEI MODULI
private var guiManager : PlayMakerFSM;        // fsm guimanager
private var moduliManager : PlayMakerFSM;     // fsm guimanager
private var nemiciManager : PlayMakerFSM;     // fsm guimanager
var moduliStart : String;                      // moduli partenza
private var globalVariables;                  // var globali
static var scenaAttuale : int;               // scena attuale
static var elencoScene : Array = ["Campagna","Periferia","Città"];

// GESTIONE PAUSE
static var isPaused : boolean = false;        // var globale
static var isFinished : boolean = false;      // var globale

// PUNTEGGI
var seconds : int;
var check : int;
var valoreBonus : int;
var plus : int;
var bonus : int;
var totale : int;

function Awake () {
    scenaAttuale = 0;
    moduliManager = GameObject.Find("__ModuliManager").
    GetComponent.<PlayMakerFSM>();
    moduliStart = moduliManager.FsmVariables.GlobalVariables.
    GetFsmString("moduliDaCaricareGlobale").Value;
    guiManager = GameObject.Find("__GUIManager").
    GetComponent.<PlayMakerFSM>();
}

function Update() {
    if(Input.GetKeyDown("escape") && !isPaused && !isFinished) {
        print("Paused");
        Time.timeScale = 0.0;
        isPaused = true;
    }
    else if(Input.GetKeyDown("escape") && isPaused &&
    !isFinished) {
        print("Unpaused");
        Time.timeScale = 1.0;
        isPaused = false;
        GameObject.Find("Camera").
        GetComponent(DepthOfFieldScatter).aperture = 7;
    }
    else if(isFinished) {
        print("Fine!");
        Time.timeScale = 0.0;
    }
}
}

```



```

function OnGUI () {
    var wButton : int = 160;           // larghezza pulsanti

    guiHurryUp = GameObject.Find("GUI_Sbrigati");
    guiMessage = GameObject.Find("GUI_Message");
    guiMessage.guiText.text = "";
    GUI.skin = guiSkin;

    // PAUSA
    if(isPaused) {

        GameObject.Find("Camera").
        GetComponent(DepthOfFieldScatter).aperture = 100;

        guiMessage.guiText.transform.position.y = 0.85;
        guiMessage.guiText.material.color = Color.white;
        guiMessage.guiText.fontSize = 35;
        guiMessage.guiText.text = "Pausa";

        if(GUI.Button (Rect((Screen.width-wButton)/2,
        (Screen.height-30)/2-60,wButton,30),
        "Continua a giocare", "button")) {
            print("Continue");
            Time.timeScale = 1.0;
            isPaused = false; GameObject.Find("Camera").
            GetComponent(DepthOfFieldScatter).aperture = 7;
        }
        if(GUI.Button (Rect((Screen.width-wButton)/2,
        (Screen.height-30)/2-20,wButton,30),
        "Reinizia il Livello", "button")) {
            print("Restart");
            Time.timeScale = 1.0;
            isPaused = false; GameObject.Find("Camera").
            GetComponent(DepthOfFieldScatter).aperture = 7;
            Application.LoadLevel("vite");
        }
        if(GUI.Button (Rect((Screen.width-wButton)/2,
        (Screen.height-30)/2+20,wButton,30),
        "Torna al Menu Principale", "button")) {
            print("Menu");
            Time.timeScale = 1.0;
            isPaused = false;
            Application.LoadLevel("start");
        }
        if(GUI.Button (Rect((Screen.width-wButton)/2,
        (Screen.height-30)/2+60,wButton,30),
        "Esci dal Gioco")) {
            print("Quit!");
            Application.Quit();
        }
    }

    // FINE GIOCO
    if(isFinished) {

        CalcolaPunteggiFinali();

        GameObject.Find("Camera").
        GetComponent(DepthOfFieldScatter).aperture = 100;
    }
}

```

```

GUI.BeginGroup(Rect(Screen.width / 2 - 125,
Screen.height / 2 - 90, 250, 180));
GUI.Box(Rect(0, 0, 250, 180), "Punteggio " +
moduliStart);
GUI.Label (Rect(20, 30, 200, 30), "Tempo Impiegato: "
+ (seconds) );
GUI.Label (Rect(20, 50, 200, 30), "Bonus Colpiti: "
+ valoreBonus);
GUI.Label (Rect(20, 70, wButton, 30), "Punteggio: "
+ totale);

// DISTRUGGO VITE
GameObject.Find("CountDownScript").
Destroy(GetComponent(_Vite));

if(GUI.Button (Rect(25,100,200,30),
"Gioca di nuovo!!", "button")) {
    print("Restart");
    GameObject.Find("CountDownScript").
    AddComponent(_Vite);
    isFinished = false;
    Time.timeScale = 1.0;
    GameObject.Find("Camera").
    GetComponent(DepthOfFieldScatter).aperture = 7;
    Application.LoadLevel("vite");
}
if(GUI.Button (Rect(25,140,200,30),
"Torna al Menu Principale", "button")) {
    print("Menu");
    isFinished = false;
    Time.timeScale = 1.0;
    Application.LoadLevel("start");
}
GUI.EndGroup ();
}
}
function CalcolaPunteggiFinali() {
    // CALCOLO PUNTEGGIO:
    seconds = _Vite.scoreSeconds;
    guiManager = GameObject.Find("__GUIManager").
    GetComponent.<PlayMakerFSM>();
    valoreBonus = guiManager.FsmVariables.
    GetFsmInt("bonusColpiti").Value;
    plus = (seconds*100/60);
    bonus = valoreBonus*10;
    totale = (seconds) + plus + bonus;
}
function ResetScena() {
    guiManager.FsmVariables.GetFsmInt("viteAttuali").Value = 3;
}

@script AddComponentMenu ("GUI/Pause GUI")

```

5.2.8 Difficoltà

L'implementazione dei livelli di difficoltà nella modalità a “tempo” avviene tramite il calcolo del numero di *checkpoint* minimo da passare e del numero di

“nemici” per ogni modulo. Viene gestita grazie all'uso della variabile “scenaAttuale” dello script “Pausa.js”. Nella modalità a “vite” invece ogni trenta secondi viene calcolato solo il numero di “nemici” da istanziare.

Codice sorgente di Difficolta.js:

```
#pragma strict

// variabili difficoltà
private var incrementoCheck : int = 5;
private var incrementoNemici : int = 1;
private var inizializzazioneCheck : int[] = [10,15,20];
private var inizializzazioneNemici : int[] = [3,4,5];
// gioco a vite
private var startTime : float;
private var intervallo : int = 30;
private var roundedSeconds : int;
var difficoltaVite : int;
// generale
var nemiciManager : PlayMakerFSM;
var tipoGioco : String;

function Awake() {

    startTime = Time.time;
    tipoGioco = Application.loadedLevelName;

    if (tipoGioco == "tempo"){
        _CountDown.difficoltaCheckPoint =
        inizializzazioneCheck[0]+
        (incrementoCheck*_Pausa.scenaAttuale);
        _RandomObject.numberToInstantiate =
        inizializzazioneNemici[0]+
        (incrementoNemici*_Pausa.scenaAttuale);
    }
    else if (tipoGioco == "vite"){
        _RandomObject.numberToInstantiate =
        inizializzazioneNemici[0]+
        (incrementoNemici*_Pausa.scenaAttuale);
    }
}

function Update() {

    var guiTime = Time.time - startTime;
    roundedSeconds = Mathf.CeilToInt(guiTime);
    if (tipoGioco == "vite"){
        if (roundedSeconds%intervallo == 0 &&
        difficoltaVite<8) {
            difficoltaVite = roundedSeconds/intervallo;
            _RandomObject.numberToInstantiate =
            inizializzazioneNemici[0]+
            (incrementoNemici*difficoltaVite);
        }
    }
}
```

5.2.9 Punteggio

Nella modalità di gioco a “tempo” vengono calcolati i punteggi parziali che devono essere salvati per ottenere il calcolo finale del punteggio totale. Nello script vengono salvati in un array e tramite la funzione “PunteggioFinale”, che viene chiamata dallo script “Pausa.js”, si avrà il conteggio finale dei punti ottenuti dall’utente.

Codice sorgente di Difficolta.js:

```
#pragma strict

private var fine : boolean = false;

static var arrayPunteggiScenari : int[] = [0,0,0];
static var punteggioFinale : int;

function PunteggioFinale () {
    if (!fine) {
        fine = true;
        for(var i:int=0; i<arrayPunteggiScenari.length; i++) {
            punteggioFinale += arrayPunteggiScenari[i];
        }
    }
    return punteggioFinale;
}
```

Capitolo 6

Il gioco in esecuzione

In questo capitolo vengono mostrati alcuni *screenshot* relativi al gioco in esecuzione per poter presentare l'effettiva realizzazione finale del progetto creato. Vengono presi in esame l'interfaccia di configurazione, il menu iniziale del gioco con la relativa interfaccia, il gioco durante l'esecuzione nella modalità a "tempo", durante l'esecuzione nella modalità a "vite" e l'interfaccia della pausa. Sono mostrate alcune scene salienti relative all'interfaccia, ai livelli costruiti tramite la modellazione ed al funzionamento dei comportamenti principali implementati durante lo sviluppo.

6.1 Configurazione

Nell'interfaccia relativa alla configurazione l'utente ha la possibilità di poter scegliere il formato della risoluzione, la qualità grafica e se giocare a tutto schermo oppure all'interno di una finestra (modalità *windowed*).

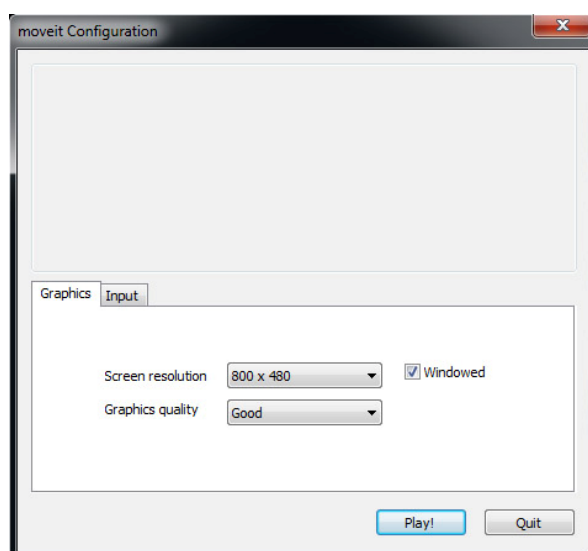


Fig. 6.1 Configurazione

6.2 Menu di Partenza

Nella schermata relativa al menu iniziale è possibile trovare tre pulsanti: “Gioco a Tempo”, “Gioco a Vite” ed “Esci”. Il primo pulsante avvia la modalità di gioco relativa, il secondo pulsante passa ad un'altra schermata con quattro pulsanti mentre l'ultimo pulsante permette di uscire dal gioco.

Nella seconda schermata della modalità di gioco a “vite” avremo a disposizione un ulteriore menu con i seguenti pulsanti: “Campagna”, “Periferia”, “Città” e “Torna al Menu”. I primi tre pulsanti avviano il gioco usando la grafica relativa mentre l'ultimo pulsante permette di tornare al menu iniziale.



Fig. 6.2 Menu Iniziale



Fig. 6.3 Menu modalità di gioco a “Vite”

6.3 Modalità di gioco a “Tempo”

La fig. 6.4 mostra l’inizio del gioco nella modalità a “tempo”. In questa fase avviene un conteggio iniziale durante il quale l’utente può prepararsi a giocare. In questa situazione inoltre è possibile notare la presenza di altri elementi dell’interfaccia grafica. In alto a destra si trova il campo di testo relativo allo scorrere del tempo, in alto sinistra si trovano i campi di testo che tengono conto del percorso e dei bonus mentre in alto al centro è possibile vedere la barra di avanzamento relativa alla quantità di percorso effettuato.

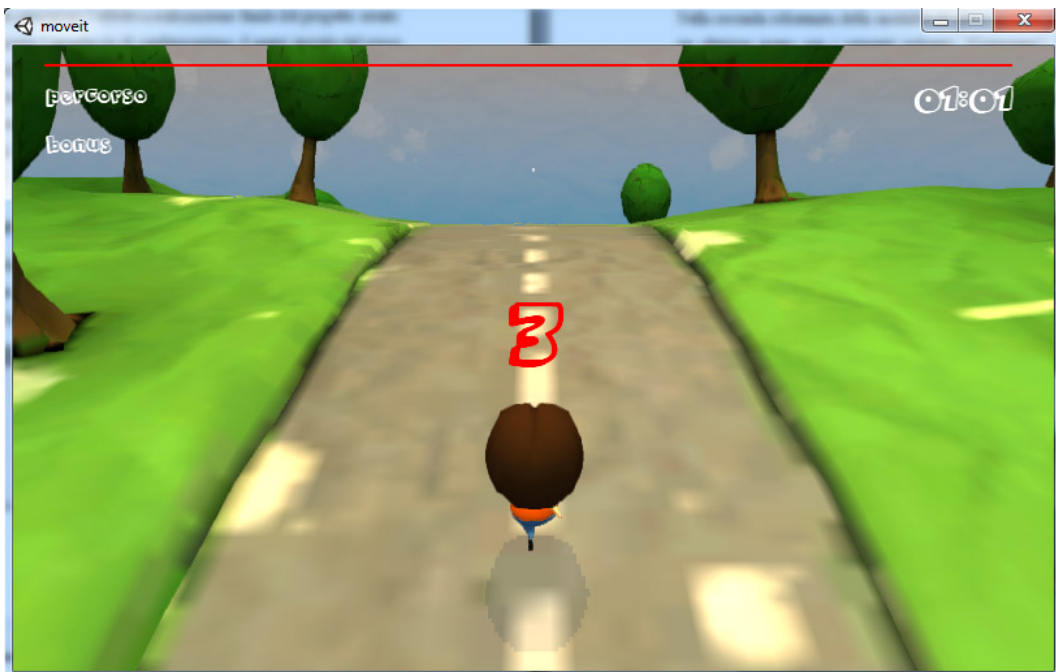


Fig. 6.4 Schermata iniziale gioco a “tempo”

Nella fig. 6.5 è possibile notare la collisione tra il personaggio ed uno degli oggetti della scena. In questo caso si nota la presenza di un animale che fa da ostacolo all’avanzamento nel percorso. Subito dopo la collisione l’oggetto colliso viene colorato di rosso applicando alla *mesh* un materiale di colore rosso con uno *shader* trasparente. Questo per far capire all’utente che se il personaggio si scontra può continuare a proseguire anche se l’ostacolo è ancora anteposto all’avanzamento. In questa modalità di gioco è importante non scontrarsi con alcun ostacolo pena la perdita di tempo utile a finire il livello.



Fig. 6.5 Collisione tra il personaggio ed un oggetto della scena

Nella fig. 6.6 viene messo in evidenza il personaggio mentre esegue l'operazione di salto di uno degli ostacoli sulla scena. In questo caso si nota il personaggio mentre sta saltando oltre al modello di uno degli animali del livello "campagna".



Fig. 6.6 Salto di uno degli ostacoli

Nella fig. 6.7 è possibile notare la pausa che intercorre tra il passaggio da un livello all'altro. In questo caso viene mostrata un'interfaccia intermedia in cui viene evidenziato la fine di un livello e tutti i punteggi relativi a quello appena finito. Attraverso il pulsante "Continua con Periferia" è possibile proseguire col prossimo livello.



Fig. 6.7 Pausa tra due livelli

Nella fig. 6.8 viene mostrata una schermata relativa al livello successivo: la "periferia". È possibile notare come cambi l'ambientazione del gioco mostrando oggetti di tipo diverso come edifici ed automobili al posto degli animali.



Fig. 6.8 Livello "periferia"

6.3 Modalità di gioco a “Vite”

La fig. 6.9 mostra l’inizio della modalità di gioco a “vite”. In questo caso vi è sempre la presenza di conteggio iniziale per dare all’utente la possibilità di prepararsi a giocare. A differenza della modalità di gioco a “tempo” è possibile notare in alto a destra la presenza delle vite del personaggio mostrate come in figura. A sinistra invece si trova lo scorrere del tempo in centesimi di secondo ed il conteggio dei bonus. In altro non vi è più la presenza della barra di avanzamento del livello infatti in questo caso non è più necessaria poichè il percorso non ha termine finchè l’utente non perde tutte e tre le vite messe a disposizione.



Fig. 6.9 Schermata iniziale gioco a “vite”

Nella fig. 6.10 è possibile notare che dopo la collisione con un ostacolo della scena viene sottratta una delle tre vite a disposizione del personaggio. Inoltre è possibile notare che dopo la collisione, oltre a colorare l’ostacolo di rosso, il personaggio diventa semitrasparente per circa tre secondi. Questo comportamento è stato implementato per dare la possibilità all’utente di non collidere con altri ostacoli subito dopo la prima collisione. In questo modo è possibile posizionarsi in un altro punto del percorso senza collidere con nessun altro oggetto e senza perdere ulteriori vite a disposizione.



Fig. 6.10 Collisione nella modalità a “vite”

6.5 Pausa

Nella fig. 6.11 è possibile notare l’interfaccia relativa al gioco mentre è nello stato di pausa. Sono stati implementati quattro pulsanti: “Continua a giocare”, “Reinizia il livello”, “Torna al menu principale” ed “Esci dal gioco”. Questo tipo di interfaccia è identica sia nella modalità a “tempo” che nella “modalità a vite”.



Fig. 6.11 Interfaccia del gioco in pausa

Bibliografia

- [1] Andries Van Dam, James D. Foley and Steven K. Feiner, *Introduction to Computer Graphics*, Addison-Wesley, agosto 1993.
- [2] Will Goldstone, *Unity 3.x Game Development Essentials*, Packt Publishing, 2 edizione, dicembre 2011.
- [3] Michelle Menard, *Game Development with Unity*, Course Technology PTR, gennaio 2011.
- [4] Randi L. Derakhshani, Dariush Derakhshani, *Autodesk 3DS Max 2012 Essentials*, Sybex Inc, giugno 2011.
- [5] Andrew Gahan, *3ds Max Modeling for Games: Insider's Guide to Game Character, Vehicle, and Environment Modeling*, Focal Press, luglio 2011.
- [6] Unity Technologies, “*Unity Manual: User Guide, FAQ, Advanced*” in <http://docs.unity3d.com/Documentation/Manual/index.html>, (ultimo aggiornamento: 13-08-2012).
- [7] HutongGames, “*PlayMaker Installation, Overview, State Machine, Editing Basics*” in: <http://hutonggames.fogbugz.com/default.asp?W10>, (ultimo aggiornamento 06-11-2011).
- [8] A. Borghese, F. Pedersini, “*Circuiti sequenziali: macchine a stati finiti*” in: http://homes.di.unimi.it/~pedersini/AER/AER09_L09.pdf, (data creazione: 2008).

Ringraziamenti

Un grazie forse è poco, ma penso sia il minimo che io possa dire per ringraziare i miei genitori per avermi sempre aiutato negli studi nonostante le difficoltà.

Un grazie a mio fratello e a tutta la mia famiglia per avermi sempre sostenuto.

Un grazie all'Ingegnere Sintoni Agide. Senza il suo aiuto in ambito didattico non penso avrei mai potuto portare a termine quest'avventura.

Un grazie ai miei più cari amici che mi hanno sempre detto di “non mollare”.

Un altro grazie va alla Professoressa Damiana Lazzaro che ha accolto l'idea di questa tesi e per la sua diponibilità.

Un grazie alla ditta Panebarco di Ravenna con la quale ho sviluppato assieme il progetto. Ringrazio Daniele, Marianna, Camilla e Matteo per la pazienza avuta e il loro supporto.

Un grazie ad Ester e ai suoi genitori per il regalo di laurea.

Un grazie a tutti coloro che ho dimenticato di citare con la fretta di scrivere queste poche righe.