ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

DIPARTIMENTO DI INFORMATICA – SCIENZA E INGEGNERIA
Corso di Laurea in Ingegneria e Scienze Informatiche

SVILUPPO IN MIXED REALITY: I DESIGN PATTERN NEI PRINCIPALI FRAMEWORK DI SVILUPPO.

Elaborato in
SISTEMI EMBEDDED E INTERNET-OF-THINGS

Relatore
Prof. ALESSANDRO RICCI

Correlatore
Dott. SAMUELE BURATTINI

Presentata da
ALEX SIROLI

Anno Accademico 2021 – 2022

Alle persone che hanno sempre creduto in me.

Indice

Introduzione

Dall'avvento dell'era dell'informazione ci siamo abituati sempre di più ad interagire con i dispositivi digitali tramite schermi in due dimensioni, con l'ausilio di mouse e tastiera nei PC o tramite le dita nei tablet e smartphone. L'obbiettivo della realtà mista è quello di portare le interazioni uomo-macchina ad un altro livello, tridimensionale e molto più immersivo.

"Ogni tecnologia sufficientemente avanzata è indistinguibile dalla magia", citazione dello scrittore Arthur C. Clarke, ci fa capire come la tecnologia abbia raggiunto un avanzamento tale da far pensare all'utente che quello che sta succedendo davanti ai suoi occhi non sia possibile. La prima volta che si vive un'esperienza in realtà mista si ha esattamente questa sensazione: di primo impatto il nostro cervello non si capacita di come sia possibile che oggetti reali e virtuali possano coesistere nello stesso ambiente in maniera tanto armoniosa, eppure è quello che sta accadendo. Negli ultimi anni abbiamo assistito ad un notevole sviluppo di questa tecnologia, grazie anche agli investimenti di importanti aziende come Microsoft, Magic Leap e Meta.

L'obbiettivo della seguente tesi è quello di analizzare quali sono ad oggi i migliori framework per lo sviluppo di software in Mixed Reality e studiare i design pattern più utili ad uno sviluppatore in questo ambito.

Nel primo capitolo vengono introdotti i concetti di realtà estesa, virtuale, aumentata e mista con le relative differenze. Inoltre vengono descritti i diversi dispositivi che consentono la realtà mista, in particolare i due visori più utilizzati: Microsoft Hololens 2 e Magic Leap 1. Nello stesso capitolo vengono presentati anche gli aspetti chiave nello sviluppo in realtà mista, cioè tutti gli elementi che consentono un'esperienza in Mixed Reality.

Nel secondo capitolo vengono descritti i framework e i kit utili per lo sviluppo di applicazioni in realtà mista multi-piattaforma. In particolare vengono introdotti i due ambienti di sviluppo più utilizzati: Unity e Unreal Engine, già esistenti e non specifici per lo sviluppo in MR ma che diventano funzionali se integrati con kit specifici come Mixed Reality ToolKit.

Nel terzo capitolo vengono trattati i design pattern, comuni o nativi per applicazioni in realtà estesa, utili per un buono sviluppo di applicazioni MR. Inoltre, vengono presi in esame alcuni dei principali pattern più utilizzati nella

programmazione ad oggetti e si verifica se e come sono implementabili correttamente su Unity in uno scenario di realtà mista. Questa analisi risulta utile per capire se l'utilizzo dei framework di sviluppo, metodo comunemente più utilizzato, comporta dei limiti nella libertà di sviluppo del programmatore.

Capitolo 1

Introduzione alla Mixed Reality

In questo primo capitolo verranno introdotti i principi fondamentali dell'eXtended Reality. Queste conoscenze di base sono necessarie per comprendere a pieno cos'è un software in realtà mista e quali sono quindi le possibili applicazioni di quest'ultimo.

Successivamente sarà illustrata nel dettaglio la Mixed Reality, esponendo quali sono i dispositivi che attualmente utilizzano questa tecnologia e quali sono gli aspetti fondamentali da considerare quando si vuole sviluppare un'applicazione in realtà mista.

1.1 Definizione di eXtended Reality (XR)

Con il termine eXtended Reality (realtà estesa) ci si riferisce a tutte le tecnologie che combinano ambienti reali e virtuali e a tutte le interazioni uomo-macchina generate dalla tecnologia.

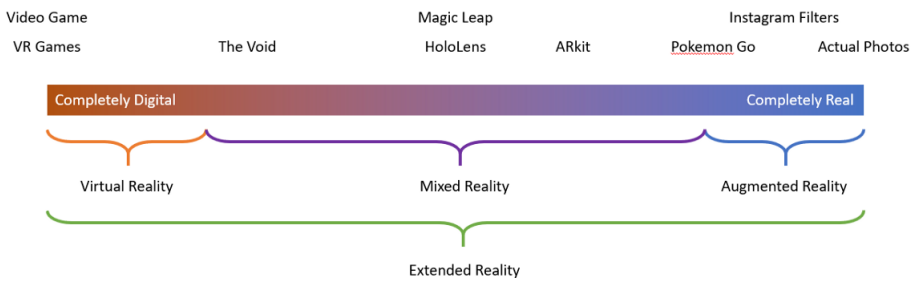


Figura 1.1: Spettro della realtà estesa [1]

L'eXtended Reality è quindi l'insieme delle tecnologie presenti e future che estendono la realtà per come la conosciamo, comprendendo l'Augmented

Reality (realtà aumentata), la Virtual Reality (realtà virtuale) e la Mixed Reality (realtà mista), come visibile in figura 1.1.

EXtended Reality viene abbreviata comunemente con l'acronimo XR, dove la "X" è una variabile che rappresenta tutte le lettere: sarà una "A" quando ci si riferisce alla realtà aumentata, una "V" quando ci si riferisce alla realtà virtuale e ad una "M" quando ci si riferisce alla realtà mista; la "R" invece rappresenta la parola Reality.

1.2 Definizioni di Realtà Aumentata e Realtà Virtuale

La realtà virtuale e la realtà aumentata, nonostante siano entrambe aree della realtà estesa, sono due tecnologie completamente diverse.

Analizzare come funzionano queste tecnologie e quali sono le differenze è quindi necessario per avere una visione d'insieme degli utilizzi anche molto diversificati che permette l'eXtended Reality.

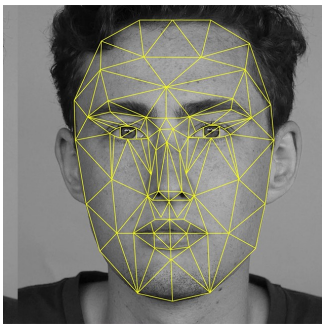
1.2.1 Augmented Reality (AR)

Con il termine Augmented Reality (realtà aumentata) ci si riferisce alla tecnologia che "aumenta" la realtà che possiamo percepire con i cinque sensi, aggiungendo degli elementi non presenti fisicamente.

L'Augmented Reality insieme alla Virtual Reality è tra le tecnologie di realtà estesa più conosciute e utilizzate. Questa diffusione è dovuta soprattutto al largo utilizzo della tecnologia da parte delle applicazioni per smartphone disponibili a tutti.

I casi più noti in cui è utilizzata la realtà aumentata sono i moderni filtri facciali disponibili sui social network, che "incollano" virtualmente una maschera al viso dell'utente (in figura 1.2a), il gioco per smartphone "Pokemon GO" che consente di visualizzare nel proprio ambiente l'animale fantastico da catturare e anche i sistemi di navigazione moderni che mostrano i percorsi da seguire direttamente sulla strada reale (in figura 1.2b).

Il punto focale degli elementi visibili con la realtà aumentata è il **posizionamento**: è possibile infatti ancorare un elemento virtuale in un punto specifico dell'ambiente fisico, ad esempio possiamo appoggiare un ologramma su una scrivania come se fosse effettivamente sopra di essa e girarci attorno per vederlo da diverse angolazioni.



(a) Esempio di maschera applicata ad un viso [2]

(b) Indicazioni stradali in realtà aumentata [3]

Figura 1.2: Applicazioni più note in realtà aumentata

Nonostante la tecnologia sia utilizzata maggiormente su smartphone esistono altri dispositivi che la sfruttano, tra questi occhiali smart, proiettori e lenti a contatto.

1.2.2 Virtual Reality (VR)

Con il termine Virtual Reality (realtà virtuale) si intende una tecnologia che simula virtualmente un ambiente reale. Con l'utilizzo di questa tecnologia, generalmente utilizzando visori che coprono completamente lo spettro visivo dell'utente (come in figura 1.3), l'ambiente percepito è completamente virtuale.

Figura 1.3: Utente che utilizza un visore VR [4]

La realtà virtuale è attualmente sfruttata principalmente nello sviluppo di videogiochi, dato che permette di navigare in ambientazioni realistiche e di interagire con gli oggetti presenti in esse in tempo reale. Le differenze rispetto alla realtà aumentata sono evidenti: la realtà virtuale crea da zero un ambiente completamente digitale, mentre la realtà aumentata sovrappone elementi digitali all'ambiente reale. La Virtual Reality è sicuramente un'esperienza più immersiva ma necessita anche di appositi visori, per questo motivo l'Augmented Reality è nettamente più diffusa.

1.3 Definizione di Realtà Mista

In questa sezione si vuole introdurre la Mixed Reality, descrivendo la definizione e analizzando le differenze con l'Augmented Reality, due tecnologie diverse ma spesso usate come sinonimi.

1.3.1 Mixed Reality (MR)

Con il termine Mixed Reality (realtà mista) si intende lo spettro compreso tra la realtà aumentata e virtuale, in cui elementi reali e virtuali coesistono in tempo reale (spettro della realtà mista visibile in figura 1.1). Con l'utilizzo di questa tecnologia è possibile muoversi in un ambiente fisico e in un ambiente virtuale allo stesso modo.

L'elemento fondamentale nella realtà mista è l'**ologramma**, cioè un elemento simulato che possiamo vedere in un ambiente fisico e che interagirà con esso. Ad esempio possiamo appendere un ologramma alla parete e manterrà quella posizione: allontanandoci si rimpicciolirà e avvicinandoci si ingrandirà, sarà possibile anche guardarlo da diverse angolazioni.

L'utente potrà anche spostare gli ologrammi liberamente nell'ambiente e anche interagire con essi per attivare specifici comportamenti, utilizzando quella che viene definita una Natural Interface.

La Natural Interface è un tipo di interazione che ci viene naturale, consente infatti di interagire con oggetti virtuali esattamente nello stesso modo in cui faremmo con gli oggetti reali. I sistemi che ne fanno uso sono quindi molto semplici da utilizzare senza bisogno di imparare un'interfaccia specifica.

1.3.2 Differenze tra Realtà Aumentata e Realtà Mista

Sia la realtà aumentata che la realtà mista mostrano l'ambiente che ci circonda con l'aggiunta di elementi virtuali, quindi potrebbero sembrare la stessa cosa. Per questo motivo spesso possono essere confuse ed utilizzate come sinonimi, quando in realtà sono due tecnologie diverse.

La differenza sta nel modo in cui viene considerato l'ambiente circostante. L'Augmented Reality riconosce le superfici come un pavimento, potremo quindi appoggiare a terra un elemento virtuale e si comporterà come se fosse effettivamente ancorato. Questa tecnologia però non riconosce altri elementi reali circostanti, per questo motivo se dovessimo spostarci dietro ad un tavolo, continueremo a vedere l'ologramma attraverso esso.

La stessa cosa capita con una maschera facciale in realtà aumentata: se dovessimo mettere un dito tra la faccia e la fotocamera, il filtro sarà visibile anche sul dito.

È quindi l'**occlusione** la differenza sostanziale tra le due tecnologie, la realtà mista riconosce gli elementi fisici circostanti e si comporta di conseguenza. Se dovessimo mettere la mano davanti ad un ologramma la parte coperta non sarà visualizzata, proprio come se fosse dietro di essa.

In figura 1.4 è visibile l'esempio dell'ologramma di un'automobile coperta parzialmente da una casa. Se l'auto rimane completamente visibile non è stata applicata correttamente l'occlusione, mentre se è visibile solo in parte l'occlusione è precisa.

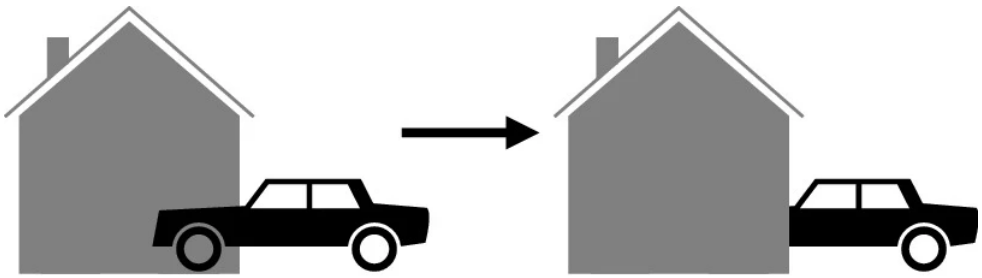


Figura 1.4: Occlusione [5]

Una corretta occlusione assottiglia ulteriormente la linea che divide gli ologrammi dell'ambiente reale. Idealmente l'utente potrebbe visualizzare degli elementi virtuali che sono stati posizionati in specifici punti della casa senza poterli riconoscere, elementi che potrebbero essere visibili solo in alcune stanze specifiche.

È evidente che questo principio sia molto importante nell'ottica di un futuro in cui ciò che è reale e ciò che è virtuale non siano più distinguibili nello stesso ambiente, ma per consentire una corretta occlusione è necessaria anche una componente hardware importante e algoritmi avanzati che analizzino questi dati in tempo utile.

Per questo motivo la realtà aumentata è utilizzabile dagli attuali smartphone in commercio, mentre ancora pochi dispositivi consentono una visualizzazione dell'ambiente in realtà mista. Il visore MR attualmente più utilizzato è Microsoft Hololens 2 (figura 1.5).

Figura 1.5: Microsoft Hololens 2 [6]

1.4 Tipologie di dispositivi MR

Le tipologie di dispositivi MR sono tre: **Head-mounted display**, **Head-up display** e **Hand-held display** (figura 1.6) [7].

(a) Head-mounted display

(b) Head-up display

(c) Hand-held display

Figura 1.6: Tipologie di dispositivi MR

1.4.1 Head-mounted display

I dispositivi Head-mounted display (HMD) sono degli strumenti indossabili, simili a dei caschetti che consentono di mostrare all'utente gli elementi in realtà mista.

Tradotto dall'inglese significa letteralmente "schermo montato sulla testa" e sono i dispositivi MR attualmente più diffusi.

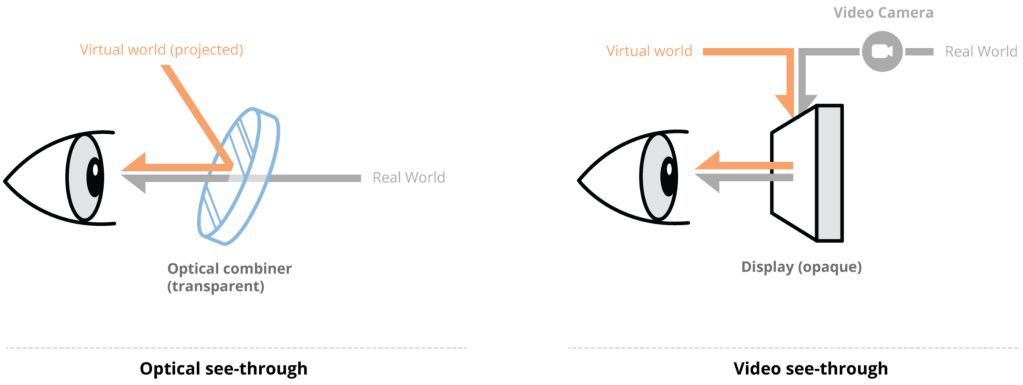


Figura 1.7: Optical see-through e Video see-through [8]

Per consentire l'esperienza in Mixed Reality possono essere utilizzate due diverse tecnologie (figura 1.7):

- **Optical see-through**: attraverso un sistema di specchi semi-trasparenti il dispositivo aggiunge gli elementi virtuali all'ambiente fisico visibile direttamente dall'utente. Non è quindi necessario riprodurre l'ambiente reale.

- **Video see-through**: il display del dispositivo copre completamente la visuale dell'utente, quindi gli ologrammi vengono inseriti nelle immagini prodotte da una telecamera che riprende l'ambiente circostante. Sarà quindi possibile vedere sia l'ambiente fisico che l'ambiente virtuale direttamente sul display, per creare la MR.

I due dispositivi MR più diffusi al momento sono Microsoft Hololens 2 e Magic Leap 1, nonostante nell'ultimo periodo siano stati annunciati nuovi dispositivi HMD: tra questi Magic Leap 2, diretto successore di Magic Leap 1 e Meta Quest Pro.

Microsoft Hololens 2

Microsoft Hololens 2 (figura 1.8) è un dispositivo head-mounted display, con tecnologia optical see-through per la realtà mista sviluppato da Microsoft con sistema operativo Windows 10, uscito nel 2019. Il dispositivo è completamente wireless, quindi non necessità di nessun cavo durante il suo utilizzo, così che l'utente possa muoversi liberamente per l'ambiente senza complicazioni. Hololens 2 è indossabile anche con gli occhiali e nonostante il peso non trascurabile di 566 grammi risulta piuttosto ergonomico, così da permetterne un utilizzo prolungato.

Figura 1.8: Microsoft Hololens 2 [6]

Entrando nel dettaglio delle caratteristiche tecniche del visore [6], dispone di 4 telecamere a luce visibile per il tracciamento della testa, 2 telecamere a raggi infrarossi per il tracciamento oculare, sensore di profondità, accelerometro, giroscopio, magnetometro e videocamera da 8 megapixel. I sensori permettono il tracciamento di entrambe le mani per la manipolazione diretta degli ologrammi, tracciamento oculare in tempo reale, tracciamento posizionale e mappatura spaziale in tempo reale. Essendo completamente wireless il dispositivo deve essere ricaricato tramite una porta USB Type-C, la durata della batteria è di 2-3 ore di utilizzo attivo.

Magic Leap 1

Magic Leap 1 (figura 1.9) è un dispositivo head-mounted display con sistema operativo Lumin OS uscito nel 2018, molto diverso da Microsoft Hololens 2. La forma e l'ergonomia sono simili a degli occhiali, dal momento che possiede due lenti separate e si appoggia anche sul naso. Non si tratta di un dispositivo completamente wireless, dato che il visore è collegato tramite cavo ad una seconda parte dalla forma rotonda che si appoggia sui pantaloni dell'utente. Diversamente dal dispositivo di Microsoft le mani non sono libere, ma viene utilizzato un controller che consente interazioni meno intuitive ma più precise.

Figura 1.9: Magic Leap 1 [9]

Entrando nel dettaglio delle caratteristiche tecniche del visore [10], dispone di una telecamera per registrare video full HD, la risoluzione del display è da 1280 x 960 pixel per occhio, integra l'accelerometro e il magnetometro, deve essere ricaricato tramite una porta USB Type-C e la durata della batteria è di circa 3.5 ore.

1.4.2 Head-up display e Hand-held display

Nonostante gli Head-mounted display siano i più utilizzati, esistono altre tipologie di dispositivi Mixed Reality: Head-up display (HUD) e Hand-held display (HHD).

Head-up display

Head-up display, letteralmente visore a testa alta, è un dispositivo originariamente sviluppato per l'aeronautica militare, in grado di proiettare informazioni utili nel campo visivo del pilota. Attualmente questi dispositivi sono utilizzati principalmente in campo automobilistico, per mostrare al guidatore informazioni sulla guida direttamente sul parabrezza (figura 1.10). La tecnologia utilizzata è optical see-through.

Figura 1.10: Dispositivo head-up display integrato in un'automobile [11]

Hand-held display

Hand-held display, letteralmente visore tenuto in mano, sono i dispositivi portatili potenti pensati per permettere esperienze in Mixed Reality come smartphone e tablet (figura 1.11). La tecnologia utilizzata è video see-through, quindi sul display sarà visualizzabile sia il mondo fisico tramite una telecamera, sia il mondo virtuale generato dal dispositivo.

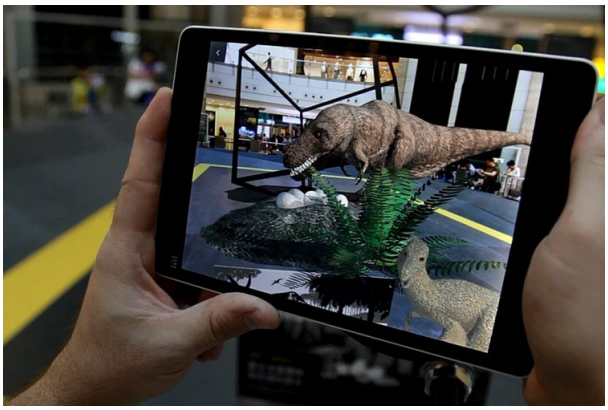


Figura 1.11: Tablet hand-held display [12]

1.5 Aspetti chiave nello sviluppo in Mixed Reality

Progettare applicazioni in realtà mista è un processo molto complesso, essendo molto diverse da tutte le altre applicazioni oggi esistenti. Non solo è necessario tenere conto delle nuove combinazioni di mondi reali e virtuali che si creano, ma anche delle nuove esperienze utente offerte [13].

Introduciamo ora gli aspetti principali per una corretta progettazione di un software in realtà mista.

1.5.1 Ologrammi

Gli ologrammi sono elementi digitali fatti di luce e suoni che appaiono nell'ambiente circostante come elementi reali [14].

Essi sono semplicemente aggiunta di luce davanti ai nostri occhi, il che significa che noi vedremo allo stesso livello sia la luce dell'ambiente reale sia la luce del display che genera l'ologramma. Questo ingannerà il nostro cervello che lo immaginerà come parte integrante dell'ambiente.

Gli ologrammi possono essere qualsiasi cosa: persone, animali, piante o qualsiasi tipo di oggetto. Possono avere anche comportamenti diversi, di tipo più realistico quando vogliamo che un elemento si integri perfettamente alla fisica a cui siamo abituati, sia di tipo fantastico. Un esempio di comportamento realistico potrebbe essere l'ologramma di una pallina, così potremmo vederla rimbalzare contro le varie superfici fisiche della stanza, creando l'illusione che sia reale. Un comportamento fantastico è invece un oggetto qualsiasi che facciamo fluttuare in aria, questo effetto potrebbe essere riprodotto per dare un effetto magico ad alcuni elementi che nel mondo reale non potremmo ricreare.

Un ologramma potrebbe anche emettere suoni, e questi verranno percepiti come se provenissero esattamente da lì. Ad esempio un ologramma di uno schermo potrebbe produrre suoni quando si visualizza un video, e l'utente muovendosi per l'ambiente continuerà a sentire il suono provenire dall'ologramma.

Mentre i suoni sono percepibili da qualsiasi punto dell'ambiente in cui ci troviamo, l'ologramma stesso è visibile solo dentro l'**holographic frame**. Gli utenti vedono tutti gli elementi in realtà mista attraverso un riquadro rettangolare in cui il visore proietta la luce. Per questo motivo gli ologrammi sono visibili solo in questo riquadro di visualizzazione, non rendendoli visibili agli estremi del nostro campo visivo, ma solo davanti [15].

Spesso capita che un elemento sia più grande dell'holographic frame, in questo caso l'ologramma verrà tagliato. L'utente per vederlo completamente potrà allontanarsi o rimpicciolirlo manualmente.

1.5.2 Interazione con gli ologrammi

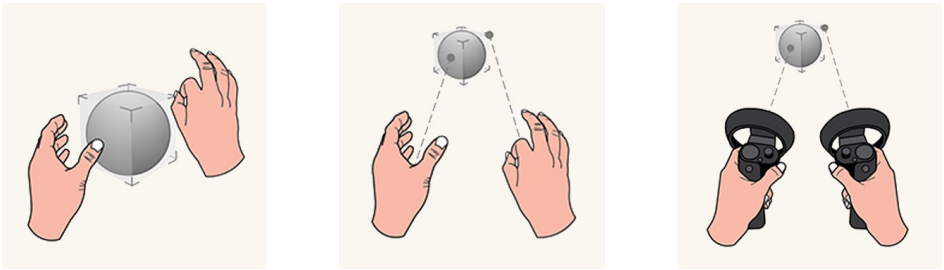
Interagire con gli ologrammi deve essere del tutto naturale. In relazione al dispositivo utilizzato potrebbe essere necessario utilizzare un controller per interagire con essi, in altri casi è sufficiente l'uso delle mani. Ad esempio Magic Leap 1 utilizza un controller da tenere in mano (figura 1.12), mentre Microsoft Hololens 2 supporta le interazioni con l'utilizzo delle mani libere.

Figura 1.12: Controller di Magic Leap 1 [16]

Tipi di interazione più complessi possono anche l'utilizzo dello sguardo o la direzione della testa, facendo un vero e proprio tracking degli occhi dell'utente in combinazione con i dati forniti dal giroscopio.

In generale, i sistemi di interazione nella realtà mista possono essere [17]:

- Mani e controller di movimento (figura 1.13) [18]:
  Con questo tipo di interazione l'utente può direttamente manipolare gli ologrammi intuitivamente.



(a) Manipolazione diretta con le mani

(b) Puntamento e commit con le mani

(c) Controller del movimento

Figura 1.13: Modalità dei controller di movimento e mani [18]

a) Manipolazione diretta con le mani: l'utente può manipolare e spostare gli ologrammi direttamente con l'uso delle mani. Esempio tipico di "Natural Interface", dal momento in cui l'interazione che abbiamo con gli elementi virtuali è esattamente lo stesso tipo di interazione a cui siamo abituati con gli elementi reali.

b) Puntamento e commit con le mani: l'utente può interagire con gli ologrammi a distanza. Questo tipo di interazione è molto simile alla manipolazione diretta con le mani ma con possibilità di inserire gli ologrammi ovunque mantenendo la possibilità di accedervi da qualsiasi distanza.

c) Controller del movimento: l'utente interagisce con gli ologrammi utilizzando un controller di precisione. Con l'uso di questi dispositivi, per una o entrambe le mani, è possibile manipolare gli ologrammi in maniera molto precisa.

- Sguardo fisso e commit [19]:

  In questo tipo di interazione l'elemento principale è lo sguardo dell'utente, senza l'utilizzo delle mani. È un metodo utilizzato principalmente per selezionare degli elementi e non per una vera e propria manipolazione degli ologrammi.

  I moderni visori MR consentono un tracking preciso dello sguardo dell'utente, pertanto con questa tecnologia sarà sufficiente fissare un elemento olografico per selezionarlo. Dal momento in cui è necessaria una precisione molto elevata del tracciamento oculare, spesso sarà necessaria una fase di calibrazione.

  Dopo la selezione effettuata con lo sguardo, è necessario un secondo input per confermare la selezione: questo processo è noto come passaggio di commit. L'input secondario può essere un comando vocale o un movimento specifico con la mano.

- Comandi vocali [20]:

  In alcuni scenari specifici l'utente può ritenere comodo o anche necessario eseguire delle azioni con il solo uso della voce. Con l'input vocale, l'utente può leggere il nome di qualsiasi pulsante ad alta voce per attivarlo e conversare con un agente digitale che può eseguire automaticamente le attività.

1.5.3 Mappatura spaziale

La mappatura spaziale è un riconoscimento dettagliato dell'ambiente circostante e consente di posizionare gli oggetti su superfici reali. Gli oggetti rimarranno ancorati all'ambiente in cui sono stati collocati. Grazie ad una corretta mappatura spaziale sarà possibile un'occlusione degli ologrammi in base ad altri ologrammi o oggetti reali molto precisa, convincendo l'utente che questi elementi siano effettivamente parte dello spazio reale [21].

Per consentire agli utenti un migliore posizionamento e spostamento degli ologrammi, è buona norma visualizzare le superfici con l'aggiunta di una griglia proiettata (figura 1.14).

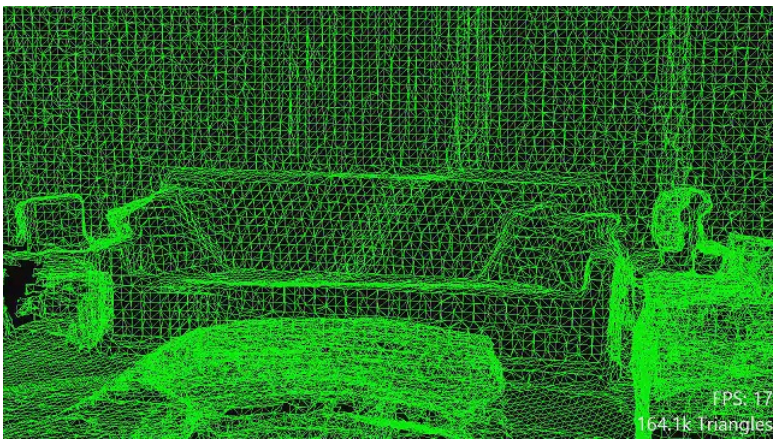


Figura 1.14: Esempio di mappatura spaziale che copre una stanza [21]

La consapevolezza spaziale è importante per diversi aspetti [7]:

- **occlusione**: permette una corretta occlusione degli oggetti, come spiegato nel dettaglio al paragrafo 1.3.2.
- **posizionamento**: permette si posizionare ologrammi su elementi reali, come ad esempio su un tavolo o su una parete.
- **persistenza**: permette agli ologrammi ancorati nell'ambiente di mantenere la giusta posizione anche dopo aver spento il dispositivo.
- **fisica**: permette agli ologrammi di comportarsi in maniera fisicamente corretta. Ad esempio l'ologramma di una pallina rimbalzerebbe correttamente contro il pavimento e le pareti dell'ambiente reale.

1.5.4 Esperienze multi-utente

Una caratteristica molto importante della realtà mista è la possibilità di gestire situazioni multi-utente.

Infatti condividere uno stesso spazio in Mixed Reality tra due o più utenti aumenta esponenzialmente i casi d'uso della tecnologia, così da vedere in tempo reale gli stessi ologrammi e anche interagire contemporaneamente con essi.

Se l'esperienza condivisa avviene tra più utenti nello stesso spazio è detta in modalità **colocated**, se invece gli utenti si trovano in luoghi separati la modalità è detta **remote** [22].

Capitolo 2

Tecnologie per lo sviluppo in Mixed Reality

In questo capitolo verranno introdotti i framework e i kit utili per lo sviluppo di un applicazione in realtà mista multi-piattaforma. Generalmente un programmatore infatti non sviluppa un'applicazione in codice nativo, perché risulterebbe un processo lento e costoso, piuttosto utilizzerà specifici framework in cui sarà comunque possibile integrare del codice per ottenere specifici comportamenti.

Essendo molto utilizzati sono considerati ad oggi uno standard per lo sviluppo in MR, e rendendo la programmazione più semplice e veloce permette a sempre più programmatori di approcciarsi alla realtà mista.

Questi ambienti di sviluppo non sono nati con l'avvento della Mixed Reality, ma sono framework già esistenti utilizzati principalmente per lo sviluppo di videogiochi 2D o 3D. Solo grazie a specifici strumenti, come MRTK, Unity e Unreal Engine diventano ambienti dedicati allo sviluppo in realtà mista. Ad oggi non esistono validi framework di sviluppo che implementano la realtà mista nativamente, ma solo ambienti già esistenti con integrazione di plug-in specifici.

2.1 Framework di sviluppo

Per sviluppare applicazioni in Mixed Reality raramente si scrive in linguaggio nativo, ma piuttosto si utilizzano strumenti di sviluppo appositi. La maggioranza dei programmatori infatti utilizza Unity o Unreal Engine per programmare questo tipo di applicazioni, per rendere lo sviluppo più semplice e veloce. Anche Microsoft nelle risorse web che mette a disposizione per i programmatori consiglia Unity per lo sviluppo di applicazioni in realtà mista per Hololens 2 [23].

2.1.1 Unity

Figura 2.1: Logo di Unity

Unity è un motore grafico multi-piattaforma sviluppato da Unity Technologies, inizialmente focalizzato allo sviluppo di videogiochi 2D e 3D, ora esteso anche all'implementazioni di software in EXtended Reality. Unity è il motore grafico più utilizzato al mondo e anche il framework di sviluppo di riferimento per Microsoft nello sviluppo di applicazioni per Microsoft Hololens 2. Fu rilasciato nel 2005 ad una conferenza Apple, ad uso esclusivo di dispositivi con sistema operativo OS X.

Generalmente si preferisce usare Unity per lo sviluppo di software perché molti aspetti di basso livello vengono gestiti direttamente dall'engine, così che anche un programmatore meno esperto possa sviluppare con solo qualche nozione di base. Per questo motivo Unity fornisce tutorial a vari livelli per imparare ad utilizzare il framework in maniera corretta [24], adatti anche a coloro che non conoscono alcun linguaggio di programmazione. Il linguaggio di riferimento per scrivere script in Unity è C#, indispensabile per fornire comportamenti specifici agli oggetti del progetto. In particolare per lo sviluppo di applicazioni in Mixed Reality, sfruttando anche funzionalità fornite da Mixed Reality ToolKit, è possibile sviluppare progetti anche di elevata complessità senza utilizzare nessun linguaggio di programmazione: anche in questo caso è possibile integrare codice in C# per aumentarne la complessità. Microsoft Learn a questo proposito propone dei corsi online introduttivi allo sviluppo di applicazioni per Microsoft Hololens 2 con Unity [23], adatti a coloro che ne conoscono almeno le funzioni base. Per questo motivo, nonostante la maggior parte degli utilizzatori siano programmatori, tutti possono approcciarsi a questa tecnologia, grazie a tutti i tutorial forniti da Microsoft e Unity ma anche grazie alla grafica semplice e intuitiva del framework. La semplicità d'uso è un grande vantaggio ma rende anche più complessa l'implementazione di funzionalità specifiche non predisposte da Unity.

Come funziona Unity?

In Unity, tutto il software si svolge nelle **scene**. Le scene sono livelli in cui si svolgono tutti gli aspetti dell'applicazione, come ad esempio i livelli di un gioco, la schermata del titolo ed i menu. Per impostazione predefinita, una nuova scena in Unity avrà un oggetto *Camera* chiamato *main camera*, eventualmente è possibile aggiungere più telecamere alla scena. La main camera esegue il rendering di tutto ciò che vede o "cattura" in una regione specifica chiamata **viewport**. Tutto ciò che è presente in questa regione diventa visibile all'utente. Nel dominio della realtà mista non sarà lo sviluppatore a posizionare la camera in base a ciò che vuole far visualizzare, ma sarà l'utente stesso che muovendosi nell'ambiente sposterà autonomamente la main camera. Generalmente questo aspetto viene gestito automaticamente da MRTK (toolkit di cui parleremo nei prossimi paragrafi), senza che se ne debba occupare il programmatore.

Una scena è fatta di oggetti, chiamati **GameObjects**. I GameObject possono essere qualsiasi cosa, dall'avatar di un utente alla GUI sullo schermo, da pulsanti a "manager" invisibili come fonti di suono. I GameObject hanno una serie di componenti collegati, che descrivono come si comportano nella scena e come interagiscono con gli altri oggetti nella scena. La proprietà più importante per qualsiasi GameObject è il componente *Trasforma*. Qualsiasi oggetto che esiste in una scena avrà una trasformazione, che ne definisce la posizione, la rotazione e la scala rispetto al mondo di gioco, o se presente al suo genitore. Possono essere gestite altre proprietà del GameObject, semplicemente selezionando il componente desiderato. È possibile anche allegare uno o più script ai GameObjects, in modo da poter dare loro un comportamento programmato [25].

2.1.2 Unreal Engine

Figura 2.2: Logo di Unreal Engine

Unreal Engine è un motore grafico multi-piattaforma sviluppato da Epic Games, attualmente è stata rilasciata la quinta generazione del motore. Il

motore grafico debutta nel 1998 con il videogioco sparatutto in prima persona *Unreal*. Come Unity permette lo sviluppo di applicazioni in Realtà Mista, nonostante sia stato pensato inizialmente per lo sviluppo di videogiochi. Attualmente è il motore principale nella visualizzazione realistica, infatti nell'ambiente grafico 3D risulta molto realistico grazie alle trame e ai materiali personalizzabili. Unreal Engine è adatto per lo sviluppo di grandi progetti, e non per la realizzazione di piccoli giochi per dispositivi mobili, nonostante abbia il supporto ad iOS e Android. Questo perché il framework è piuttosto complesso e difficile da imparare, a differenza di Unity. Nonostante la complessità piuttosto elevata, Unreal Engine utilizza un sistema di scripting visivo chiamato *Blueprint* simile a un diagramma di flusso, che espone le istruzioni in una forma facile da seguire. Per fornire comportamenti specifici agli oggetti è possibile integrare codice in C++, dando grande controllo al programmatore sull'intero sistema. Essendo il motore open source, permette agli sviluppatori più esperti di modificare o aggiornare il codice sorgente in qualsiasi momento. Unreal Engine ha il vantaggio di offrire molti effetti di post-produzione, veloci e con molte funzionalità. Per quanto riguarda lo sviluppo in EXtended Reality è considerato il migliore motore per creare ambientazioni virtuali immersive, grazie all'eccellente livello di prestazioni. Essendo più complesso, non dispone di una community molto estesa, e la documentazione risulta debole in alcuni punti [26].

2.2 OpenXR

Figura 2.3: Logo di OpenXR

OpenXR è un'Application Program Interface (API) ad alte prestazioni royalty-free creato da Khronos per piattaforme e dispositivi di EXtended Reality, quindi Augmented (AR), Virtual (VR) e Mixed Reality (MR). Khronos Group è un consorzio fondato nel 2010 con lo scopo di sviluppare API libere da royalty per un'ampia varietà di dispositivi. Altre API note del gruppo sono OpenGL, OpenVX e Vulkan. L'idea di creare OpenXR nasce con il rilascio dei primi visori per la realtà virtuale: Oculus Quest e HTC Vive. Ognuno dei visori presentava un SDK (Software Development Kit) specifico per lo sviluppo di applicazioni, quindi nonostante questi visori presentassero funzionalità simili

come la gestione della pressione di un pulsante, era necessario programmare due applicazioni differenti. Questo comportava quindi un maggior dispendio di tempo per lo sviluppo delle applicazioni ma anche un incremento dei costi, problemi destinati a peggiorare con il rilascio di nuovi visori con SDK specifici. Molte funzioni sono comuni tra visori diversi ma anche tra tecnologie diverse, ad esempio sia dispositivi VR che MR devono conoscere dove sono le mani nel mondo reale. Khronos Group si pone come obbiettivo quello di risolvere il problema sopra-citato, detto **frammentazione**, con la creazione di un'unica API comune a tutti, detta OpenXR (figura 2.4). Con l'utilizzo di OpenXR sarà possibile scrivere il codice una volta sola per un'ampia gamma di dispositivi diversi.

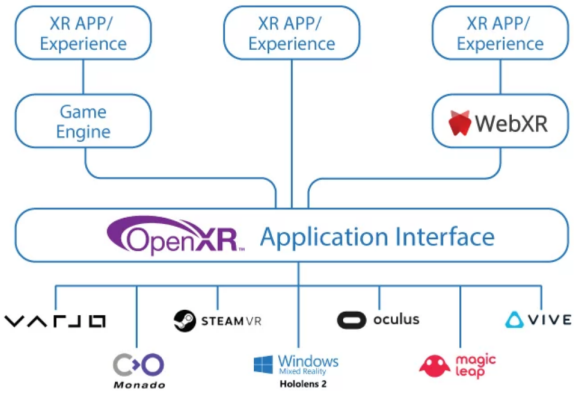


Figura 2.4: Posizionamento di OpenXR [27]

Dal rilascio OpenXR è ampiamente utilizzato, rendendo possibile la portabilità su dispositivi che utilizzano l'API. La figura 2.5 raggruppa alcune delle aziende che supportano pubblicamente OpenXR.

Figura 2.5: Alcune aziende partner di OpenXR [28]

Nonostante OpenXR, il problema della frammentazione non è completamente risolto. Ad esempio, i due dispositivi di Mixed Reality attualmente più utilizzati: Microsoft Hololens 2 e Magic Leap 1 utilizzano OpenXR specifici per il dispositivo, rendendo la portabilità della stessa applicazione da un visore all'altro senza apportare modifiche impossibile.

2.3 Mixed Reality ToolKit (MRTK)

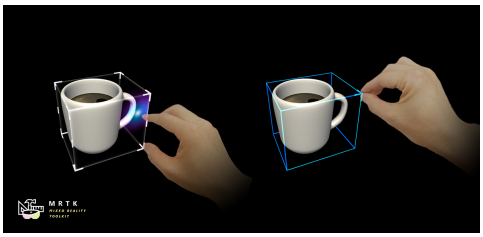
Figura 2.6: Logo di Mixed Reality ToolKit

Mixed Reality ToolKit (MRTK) è un kit di sviluppo realizzato da Microsoft open-source e multi-piattaforma [29]. MRTK fornisce al programmatore una raccolta di componenti e funzionalità per la realizzazione di applicazioni in Mixed Reality per Unity e Unreal Engine a più dispositivi possibili.

Alcune delle funzioni che Mixed Reality ToolKit mette a disposizione sono:

- Fornisce un sistema di input adatto a diverse piattaforme MR e blocchi predefiniti per le interazioni con l'interfaccia utente (figura 2.7).
- Permette una rapida creazione di prototipi tramite simulazione per visualizzare in tempo reale le modifiche che si stanno effettuando.
- Tramite modularità rende il progetto in realtà mista più leggero.

(a) Controllo pulsante che supporta vari metodi di input, in questo caso interazione con la mano



(b) Interfaccia utente standard per la modifica di oggetti nello spazio 3D

Figura 2.7: Esempi di blocchi predefiniti dell'esperienza utente [29]

Un importante vantaggio di MRTK è la **modularità**: non è necessario in ogni progetto importare tutte le funzionalità che il kit mette a disposizione, ma lo sviluppatore può liberamente scegliere solo quelle che reputa utili per lo sviluppo. Questa caratteristica permette quindi alle applicazioni in realtà mista sviluppate con Mixed Reality ToolKit di mantenere dimensioni accettabili.

Un altro importante vantaggio è la **configurabilità**: i programmatori hanno la possibilità di sostituire alcune componenti fornite da MRTK con delle proprie, per implementare funzionalità specifiche non presenti nel kit.

MRTK include il supporto a più piattaforme, in particolare supporta ufficialmente OpenXR, permettendo di utilizzare entrambi i plug-in contemporaneamente per consentire uno sviluppo di applicazioni XR più rapido, semplice e multi-piattaforma.

Microsoft mette a disposizione delle dispense online per approcciarsi correttamente a MRTK e spiegando nel dettaglio come creare un'applicazione in realtà mista per Hololens 2 con l'utilizzo del kit.

Capitolo 3

Design Pattern nello Sviluppo di Applicazioni di Realtà Mista

I design pattern sono fondamentali nella programmazione ad oggetti e sviluppando applicazioni in codice nativo ne abbiamo piena libertà di utilizzo.

Spesso però i software in realtà mista non sono scritti in codice nativo, ma si utilizzano framework di sviluppo come Unity e Unreal Engine. Questi framework rendono lo sviluppo più semplice e di alto livello, ma questa semplicità comporta delle limitazioni nelle scelte del programmatore.

L'obbiettivo di questo capitolo è quello sia di descrivere i nuovi design pattern specifici per lo sviluppo in realtà mista sia di analizzare il trade-off tra semplicità e libertà di sviluppo che ci offrono i framework di sviluppo, in particolare capire se è possibile implementare i design pattern più comuni. L'ambiente di sviluppo preso di riferimento per studiare l'implementazione dei vari pattern è Unity.

La seguente tabella mostra tutti i pattern che tratteremo, divisi tra desgin pattern per lo sviluppo in realtà mista e i principali design pattern di Unity.

Design pattern per lo sviluppo in Realtà Mista	**I principali design pattern su Unity**
Scene Graph Node pattern	Game Loop
Scripting pattern	State Machine
Scene Graph pattern	Factory
	Singleton
	Command
	Model-View-Controller

3.1 I design pattern

Progettare un software è un'attività molto complessa. Capire cosa sono i design pattern e come vengono utilizzati è indispensabile per la buona progettazione di un programma.

3.1.1 Introduzione ai design pattern

Gli sviluppatori in fase di progettazione si ritrovano spesso a risolvere problemi comuni, i design pattern sono metodologie convenienti che li risolvono. L'idea dietro ai design pattern è semplice: catalogare interazioni comuni tra oggetti di un software che i programmatori hanno spesso trovato utili [30].

"Don't reinvent the wheel": nessun programmatore ha mai pensato di reimplementare da capo una libreria che funziona, per lo stesso motivo è comodo utilizzare una soluzione progettuale generale ad un problema ricorrente, cioè un pattern [31].

Utilizzare uno o più design pattern comporta anche i seguenti vantaggi:

- ogni pattern può causare dei problemi noti: conoscerli in anticipo renderà più veloce la risoluzione, e si farà meno fatica a correggere problemi che hanno una struttura simile.
- utilizzare i pattern rende il progetto più chiaro: spesso rappresentano un "vocabolario" comune tra i programmatori, questo renderà più semplice e chiara la comunicazione e la lettura del codice.
- la qualità del progetto aumenta: un corretto uso dei pattern rende il progetto più snello e favorisce il riuso di codice e la sua manutenzione.

3.1.2 Classificazione dei design pattern

Possono essere classificati in base al loro scopo (**purpose**):

- **creazionali**: riguardano la creazione degli oggetti.
- **strutturali**: definiscono la struttura e la composizione di classi e oggetti, utilizzando concetti di ereditarietà e polimorfismo.
- **comportamentali**: riguardano l'interazione e la distribuzione di responsabilità tra classi e oggetti.

Possono anche essere classificati in base al raggio d'azione (**scope**):

- **classi**: pattern che riguardano le relazioni tra classi e sottoclassi, generalmente sono basate sul concetto di ereditarietà quindi tratta aspetti statici (compile-time).

- **oggetti**: pattern che riguardano le relazioni tra oggetti, che potendo cambiare durante l'esecuzione tratta aspetti dinamici (run-time).

Alcuni pattern non operano a livello di design del sistema, per questo motivo non possono essere considerati propriamente design pattern. Ad esempio i pattern architetturali sono un livello più ampio rispetto ai design pattern, ed indicano l'organizzazione strutturale di tutto il sistema. Esempio noto di pattern architetturale è Model-View-Controller (MVC).

3.2 Design pattern per lo sviluppo in Realtà Mista

In questa sezione si vogliono illustrare i design pattern più utili nella realizzazione di un'applicazione in Mixed Reality.

Questo tipo di applicazioni presentano problemi ricorrenti da risolvere, per questo motivo è importante trovare soluzioni comuni e standard per gestirli. Questo nuovo dominio di sviluppo ha dato vita a nuovi design pattern nativi per lo sviluppo di software in realtà estesa, utili allo sviluppo in realtà mista ma anche per la realtà aumentata e virtuale.

Essendo il campo della realtà estesa recente e in forte sviluppo, verranno creati nuovi design pattern specifici per risolvere problemi specifici. Mentre ora i pattern sono generici per software in realtà estesa, dal momento in cui condividono alcuni aspetti di programmazione, in futuro potrebbero nascere design pattern che risolvono problemi della singola tecnologia. Col tempo avremo quindi pattern dedicati alla Mixed Reality, diversi sia da quelli di realtà aumentata sia virtuale.

3.2.1 Scene Graph Node pattern

Scene Graph Node è un pattern nato con lo sviluppo in realtà estesa e riguarda il sotto-sistema del software che contiene tutto il codice specifico dell'applicazione [32].

Qual è l'obbiettivo?

L'obbietti del pattern è quello di incorporare correttamente il modello virtuale con il mondo fisico.

Perché è stato ideato?

Nella realtà mista, l'interazione dell'utente è strettamente collegata all'ambiente fisico. Di conseguenza, le applicazioni sono spesso connesse a luoghi del mondo reale. Con questo pattern, l'applicazione è perfettamente integrata nell'ambiente.

Che struttura ha?

Il mondo fisico attorno all'utente viene modellato come un albero di nodi (figura 3.1).

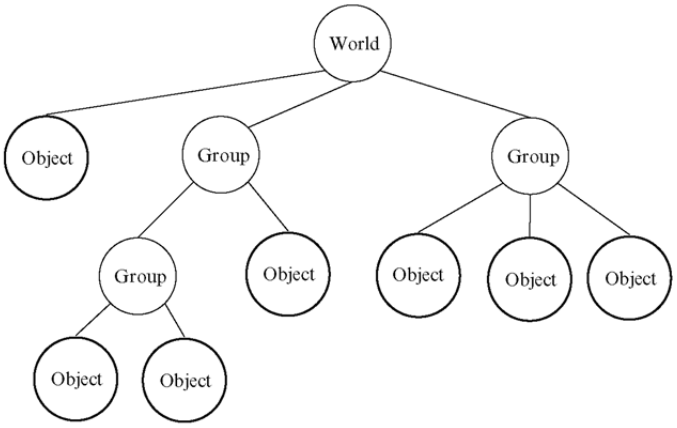


Figura 3.1: Esempio di albero di nodi [33]

Ogni nodo può essere di qualsiasi tipo, generalmente vengono utilizzati oggetti grafici come delle sfere. Sono comunque presenti anche oggetti non grafici che comprendono codice di controllo. Ogni nodo presente nell'albero può possedere molti figli ma un singolo padre i quali effetti vengono propagati a tutti i nodi figli sottostanti. Un'operazione effettuata ad un singolo nodo provoca la modifica a tutti i figli di esso. Con l'utilizzo di questo pattern è possibile gestire anche ambienti condivisi tra più utenti.

Che vantaggi e svantaggi comporta?

L'approccio basato sugli alberi di nodi è un modo semplice e naturale che consente di avere effetti su un intero gruppo di nodi con la stessa facilità con cui si manipola un singolo oggetto.

3.2.2 Scripting pattern

Scripting è un pattern nato con lo sviluppo in realtà estesa e riguarda il sotto-sistema del software che contiene tutto il codice specifico dell'applicazione [32].

Qual è l'obbiettivo?

Sviluppare rapidamente nuove applicazioni.

Perché è stato ideato?

I vincoli di prestazioni in tempo reale di un'applicazione generalmente non sono molto forti, quindi è possibile sviluppare velocemente nuove applicazioni con l'utilizzo di un linguaggio di programmazione compilato.

Che struttura ha?

Durante lo sviluppo dell'applicazione, si esegue un *wrapping* di scripting di tutti i componenti che hanno vincoli di prestazioni, questi script sono scritti in linguaggi compilati come ad esempio C++.

Che vantaggi e svantaggi comporta?

Lo sviluppo di applicazioni con questo tipo di script consente una prototipazione rapida, ma richiede componenti potenti che implementano funzionalità importanti. Lo svantaggio è che l'approccio di scripting non è adatto per applicazioni molto complesse.

3.2.3 Scene Graph pattern

Scene Graph è un pattern nato con lo sviluppo in realtà estesa e riguarda il sotto-sistema del software che gestisce la visualizzazione degli elementi 3D e 2D (come testo inserito dell'ambiente) e la riproduzione audio percepibili dall'utente [32].

Qual è l'obbiettivo?

Utilizzare un componente di rendering che consenta scene più complesse e dinamiche.

Perché è stato ideato?

Per la rappresentazione di ambienti 3D, i *Scene Graph* si sono rivelati una scelta ragionevole. Il livello di astrazione è superiore a quello di OpenGL (Open Graphics Library), API per più linguaggi multi-piattaforma divenuto uno standard per la produzione di grafica 3D, e risultano molto più potenti e flessibili dei file VRML (Virtual Reality Modeling Language), formato di file per la rappresentazione grafica 3D online che possiede un'interfaccia di sviluppo dell'applicazione limitata.

Che struttura ha?

Questo pattern utilizza gli stessi alberi utilizzati per il pattern Scene Graph Node al paragrafo 3.2.1, ma i due hanno diversi obbiettivi. Scene Graph Pattern infatti si occupa della parte di rendering del software, quindi del suo output. Utilizzando la stessa struttura i due pattern sono utilizzati insieme.

Un *Scene Graph* viene usato se non si ha bisogno della flessibilità e dell'accesso alla grafica di basso livello forniti da OpenGL, ma si vuole comunque eseguire il rendering di scene più complesse e si ha bisogno di un accesso più dinamico di quello offerto da un VRML.

Che vantaggi e svantaggi comporta?

Consente il rendering di scene complesse ma può limitare le possibilità di modellazione dell'applicazione.

3.3 I principali design pattern su Unity

In questa sezione si vogliono prendere in esame alcuni dei pattern più utilizzati nello sviluppo ad oggetti, e verificare se e come possono essere implementati in Unity per la programmazione di progetti in realtà mista.

L'obbiettivo è quello di verificare la flessibilità dello sviluppo con Unity, anche per quanto riguarda la libertà di scelta dei design pattern che il programmatore vuole utilizzare.

3.3.1 Game Loop

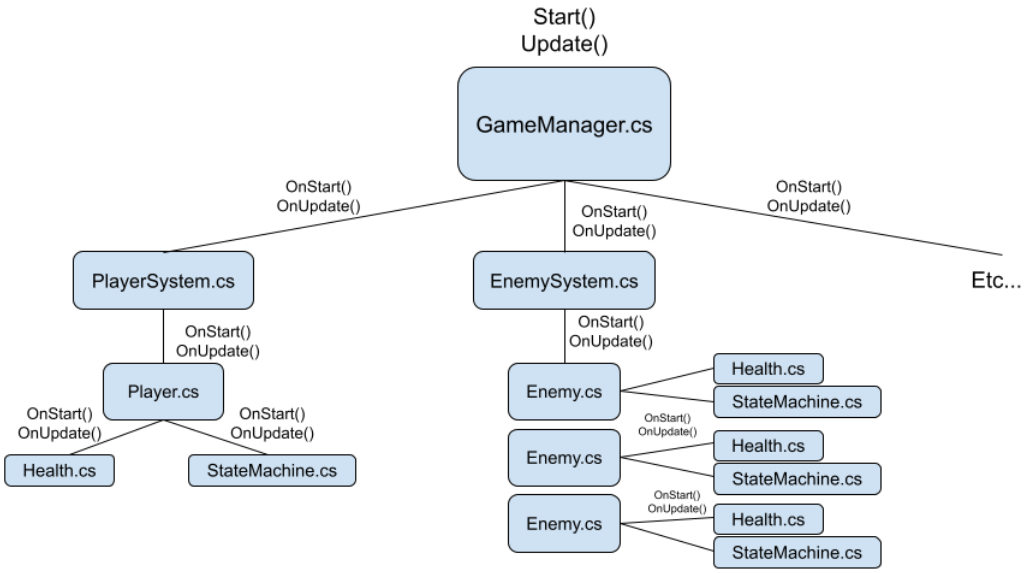


Figura 3.2: UML di un esempio di utilizzo di *Game Loop* [34]

Descrizione del pattern

Il design pattern *Game Loop* separa la progressione del tempo di gioco dall'input dell'utente e dalla velocità del processore.

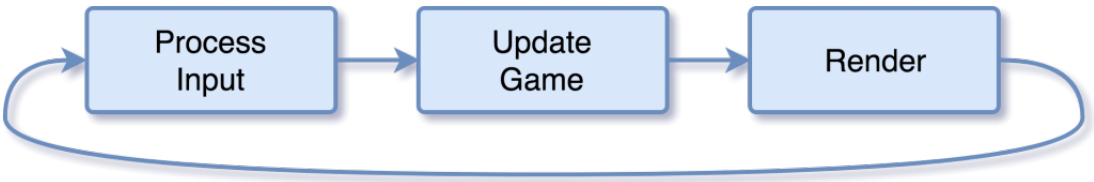


Figura 3.3: Fasi del *Game Loop*. [35]

In altre parole, il modello Game Loop assicura che il tempo di gioco progredisca alla stessa velocità in tutti i diversi hardware e configurazioni. Il game loop viene eseguito continuamente durante il funzionamento dell'applicazione, e ad ogni ciclo viene processato l'input da parte dell'utente, aggiornato lo stato del programma e renderizzato l'ambiente [36], come visibile in figura 3.3.

Nonostante il nome, questo pattern non è funzionale solo allo sviluppo di videogiochi, ma anche ad altri tipi di applicazioni. Infatti questo design pattern è leggermente diverso da quelli che vedremo successivamente, perché per certi software (come quelli in realtà estesa) e quasi tutti i giochi la sua implementazione è indispensabile per il funzionamento degli stessi.

Data questa particolarità, i game engine come Unity e Unreal Engine implementano il game loop nativamente, senza bisogno che il programmatore lo sviluppi esplicitamente.

Spesso il game loop è proprio ciò che fa la differenza tra un engine e una libreria: con le librerie il programmatore sviluppa in loop utilizzando le funzioni messe a disposizione dalle librerie, l'engine invece costruisce il loop utilizzando il codice del programmatore [36].

Utilizzando questo pattern il programma non rimane in attesa degli input dell'utente per funzionare, ma gira continuamente senza aspettare. Gli input dell'utente vengono elaborati ma il software non rimane mai in attesa: se ad esempio l'utente di un videogioco rimane fermo a fissare lo schermo gli effetti visivi continueranno a muoversi, il sottofondo musicale sarà ancora udibile e se è sfortunato i nemici colpiranno il suo avatar [37].

Questa è la parte principale di un game loop, ed è facilmente illustrabile in questo modo:

```
while (true)
{
  processInput();
  update();
  render();
}
```

Dato che il loop non si ferma mai è importante capire quanto velocemente esegue un ciclo. Se misuriamo il valore di questa velocità in relazione al tempo reale, avremo i "frame al secondo", in inglese "frames per second" (FPS). Più alto è questo valore, più aggiornamenti grafici verranno effettuati, di conseguenza sarà evidente una maggiore fluidità del programma. Allo stesso modo pochi FPS produrranno un programma lento simile ad un filmato in stop-motion. Per sapere a quanti frame al secondo verrà eseguito un programma

bisogna considerare sia quanto lavoro bisogna eseguire ad ogni ciclo, sia la potenza hardware della macchina su cui viene lanciato il software [36].

Main Loop di Unity

Unity implementa implicitamente un game loop per conto del programmatore, di cui non deve occuparsi. Questo loop su Unity è chiamato **Main Loop** e non è visibile dallo sviluppatore. Come tutti i game loop, il suo compito è quello di raccogliere input, aggiornare gli oggetti del software ed eseguire il rendering.

Generalmente quando si sviluppa un programma con Unity si creano diverse classi di tipo MonoBehaviour, queste verranno inizializzate e aggiornate in modo indipendente. Le procedure da eseguire in questi due casi sono implementate nel metodo *start()* per quanto riguarda l'inizializzazione, e nel metodo *update()* per quanto riguarda gli aggiornamenti.

In figura 3.4 sono visibili nel dettaglio tutti i passaggi svolti dal main loop, consultabile nella documentazione ufficiale di Unity.

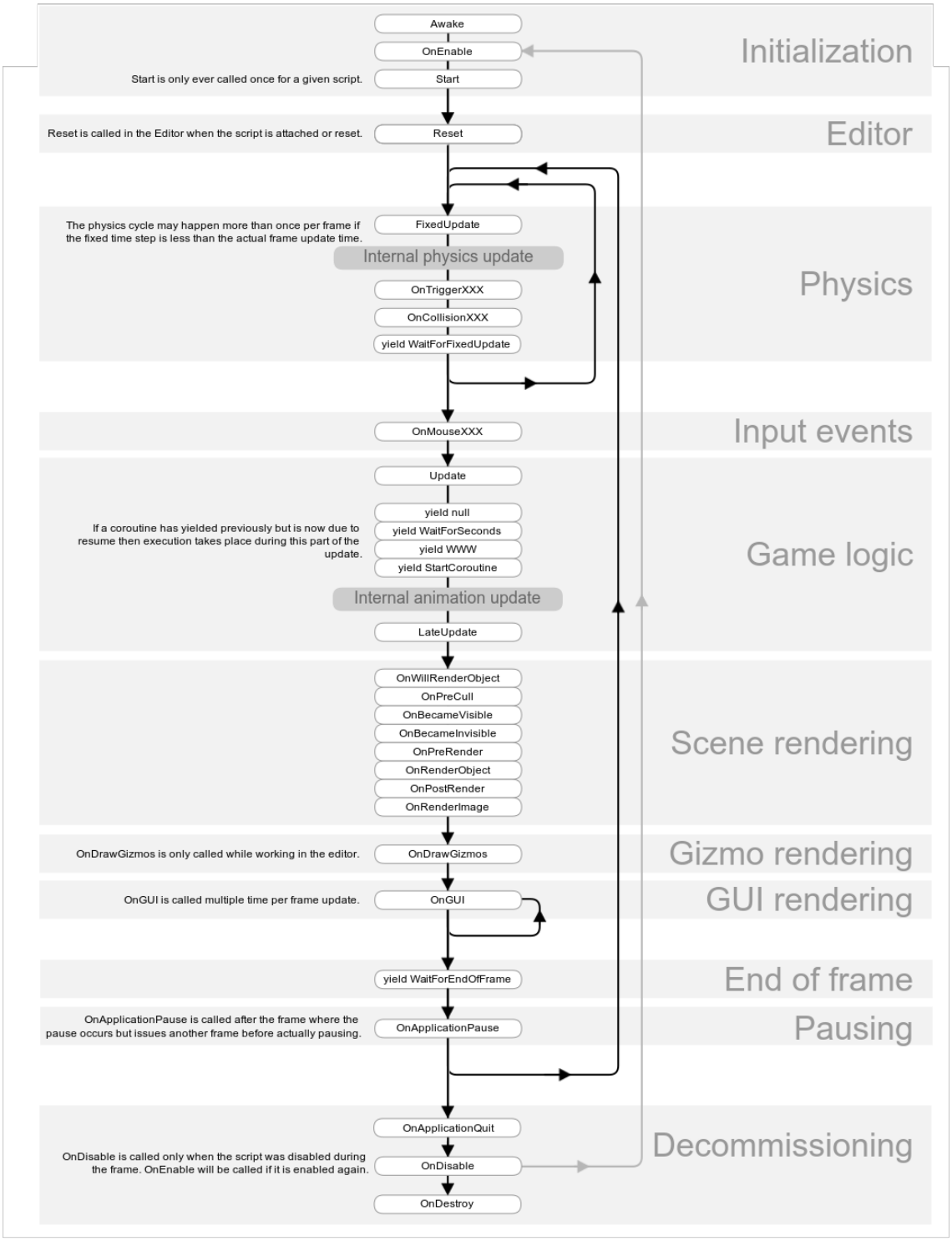


Figura 3.4: Main Loop di Unity [38]

3.3.2 State Machine

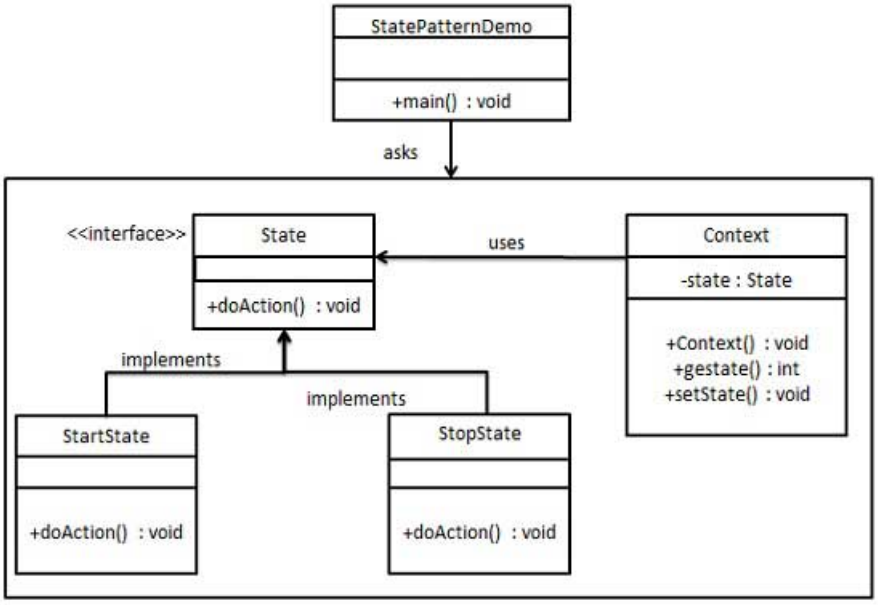


Figura 3.5: UML di un esempio di utilizzo di *State Machine*. [39]

Descrizione del pattern

State machine è un design pattern di tipo comportamentale, che ha come raggio d'azione gli oggetti.

In italiano "macchina a stati", è uno dei design pattern più utilizzati per lo sviluppo di videogiochi. Come è intuibile dal nome, *state machine* è appunto una "macchina" che contiene diversi stati. Ad esempio un videogioco potrebbe avere un "Menu State" quando si accede al menù, un "Game State" mentre si sta giocando ed un "GameOver State" quando è finita una partita. Nell'ambiente della realtà mista potremmo avere diversi stati per un ologramma, ad esempio mentre lo stiamo spostando, ridimensionando e semplicemente quando è fermo. In ciascuno di questi stati l'ologramma potrebbe avere diversi comportamenti o animazioni [40].

Vantaggi

State machine oltre a rendere il codice del software molto più comprensibile e intuitivo, riduce al minimo la complessità condizionale: non saranno quindi necessari istruzioni if e switch per assegnare comportamenti specifici agli oggetti. Sarà molto più semplice individuare i comportamenti specifici

per ogni stato e semplicemente effettuare una transizione da uno stato ad un altro quando necessario [41].

Svantaggi

State Machine richiede la scrittura di molto codice. In base a quanti sono gli stati possibili e i vari metodi di transizione si può arrivare a scrivere diverse dozzine di metodi [41].

Implementazione per sviluppo in realtà mista su Unity

Il pattern *State Machine* può risultare molto utile nello sviluppo di software in realtà mista.

Nell'ambito MR potrebbe risultare comodo assegnare degli stati agli ologrammi, ad esempio quando sono in movimento, mentre vengono ridimensionati o mentre sono fermi nell'ambiente. Il pattern consentirebbe così di assegnare comportamenti specifici all'ologramma in ciascuno di questi stati.

Utilizzando Unity è possibile implementare in maniera semplice *State Machine*, implementando in maniera corretta sia la classe di cui vogliamo creare gli stati sia gli stati stessi. La classe di cui vogliamo poter cambiare stato necessita del campo in cui vi è il riferimento allo stato corrente e due metodi fondamentali.

```
using UnityEngine;

// Uses BaseState as base class for storing current state.
public class StateMachine : MonoBehaviour
{
    // Reference to currently operating state.
    private BaseState currentState;

    private void Update()
    { ... }

    public void ChangeState(BaseState newState)
    { ... }
}
```

L'implementazione dei due metodi è la seguente:

- **Update**: che consente di fare l'aggiornamento dello stato corrente.

```
// Unity method called each frame
private void Update()
{
    if (currentState != null)
    {
        currentState.UpdateState();
    }
}
```

- **ChangeState**: che consente la transizione da uno stato ad un altro.

```
// Method used to change state
public void ChangeState(BaseState newState)
{
    // If we currently have state, we need to destroy it!
    if (currentState != null)
    {
        currentState.DestroyState();
    }

    // Swap reference
    currentState = newState;

    // If we passed reference to new state, we should
    // assign owner of that state and initialize it!
    // If we decided to pass null as new state, nothing
    // will happened.
    if (currentState != null)
    {
        currentState.owner = this;
        currentState.PrepareState();
    }
}
```

Le classi "stato" invece necessitano del campo che contiene il riferimento alla macchina a stati che li possiedono e tre metodi fondamentali:

- **PrepareState**: da eseguire all'avvio dello stato.
- **UpdateState**: che aggiorna lo stato.
- **DestroyState**: da eseguire alla distruzione dello stato.

```
//This is base state script implementation.
public abstract class BaseState
{
    // Reference to our state machine.
    public StateMachine owner;

    // Method called to prepare state to operate
    public virtual void PrepareState()
    { ... }

    // Method called to update state on every frame
    public virtual void UpdateState()
    { ... }

    // Method called to destroy state
    public virtual void DestroyState()
    { ... }
}
```

3.3.3 Factory

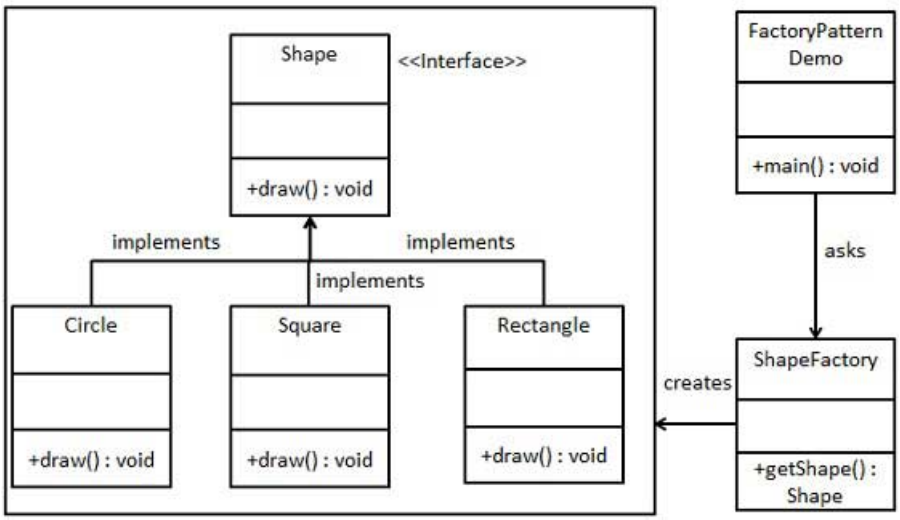


Figura 3.6: UML di un esempio di utilizzo di *Factory* [42]

Descrizione del pattern

Factory è un design pattern di tipo creazionale, che ha come raggio d'azione gli oggetti.

Costruire correttamente oggetti è una responsabilità spesso non banale, per questo motivo può essere meglio scorporarla dalle classi stesse. Una *factory*, cioè una "fabbrica" di oggetti, è una classe che attraverso i suoi metodi permette di creare oggetti di un certo tipo o interfaccia.

Vantaggi

Usare *factory* può risultare molto utile per gestire la creazione di un oggetto piuttosto complesso o per creare più oggetti simili tra loro così da poter riutilizzare codice nella classe. In alcuni casi è troppo complicato generare l'istanza di un oggetto con il solo utilizzo di "new", quindi *factory* garantisce il riutilizzo e l'uniformità del codice riducendo al minimo i potenziali errori.

Svantaggi

Se si considera lo sviluppo di un software molto ampio, l'abuso di *factories* potrebbe comportare una riduzione delle prestazioni. Per quanto riguarda il design pattern *factory* gli svantaggi sono trascurabili in relazione ai benefici visti in precedenza [43].

Implementazione per sviluppo in realtà mista su Unity

Il pattern *Factory* può essere utile nello sviluppo in realtà mista quando si vogliono generare numerosi ologrammi dello stesso tipo. Utilizzando una factory sarà possibile riferirsi ad un ologramma pre-impostato e generare qualsiasi numero di istanze facilmente.

Creare una factory con Unity è piuttosto semplice, sarà necessario un campo con il riferimento all'ologramma prefabbricato (*prefab* su Unity) ed un metodo che fornisca nuove istanze dell'ologramma.

Una classe factory semplice su Unity avrà quindi questa struttura:

```
// Factory design pattern.
public class Factory : MonoBehaviour
{
    // Reference to prefab.
    private MonoBehaviour prefab;

    // Creating new instance of prefab.
    public MonoBehaviour GetNewInstance()
    {
        return Instantiate(prefab);
    }
}
```

Se si desidera una factory generica, che quindi generi oggetti anche di tipo diverso, è possibile implementarla in questo modo:

```
/// Generic Factory design pattern
public class GenericFactory<T> : MonoBehaviour
    where T : MonoBehaviour
{
    // Reference to prefab of whatever type.
    private T prefab;

    // Creating new instance of prefab.
    public T GetNewInstance()
    {
        return Instantiate(prefab);
    }
}
```

3.3.4 Singleton

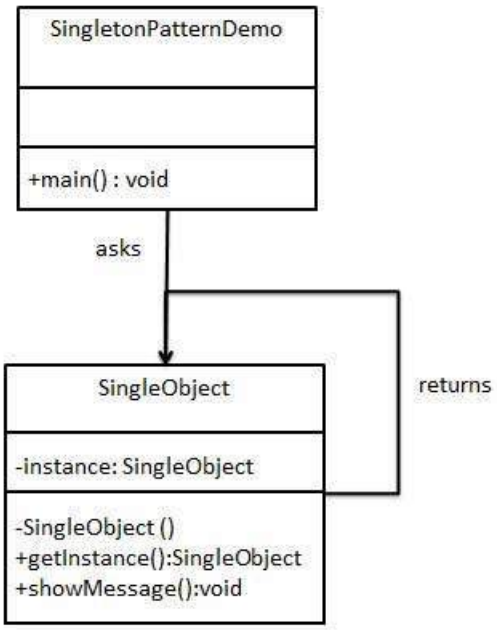


Figura 3.7: UML di un esempio di utilizzo di *Singleton* [44]

Descrizione del pattern

Singleton è un design pattern di tipo creazionale, che ha come raggio d'azione gli oggetti.

L'intento è garantire che una classe abbia un'unica istanza, accessibile facilmente e globalmente da diverse classi, senza preoccuparsi di fornirne il riferimento.

Le caratteristiche della classe *singleton* sono:

- La classe è responsabile di tenere traccia di tale unica stanza.
- La classe deve impedire la creazione di altri oggetti della classe.
- La classe deve fornire l'accesso all'oggetto staticamente.

Vantaggi

Il vantaggio principale di *singleton* è la certezza che esista una sola istanza della classe, utile per i sistemi responsabili della localizzazione, gestione dei file o della rete. Anche il codice risulterà più leggero, dal momento in cui

non è più necessario portare i riferimenti all'oggetto in tutte le classi che lo usano. Con *singleton* è possibile gestire la creazione dell'oggetto al bisogno (lazy singleton).

Svantaggi

Singleton è un design pattern da usare con cautela, perché creare diversi *singleton* con l'aumentare della complessità del software comportano un effetto contrario sulla semplicità di sviluppo. Anche la manutenzione dello stesso risulterebbe più complessa e tornare indietro dalla scelta di usare il *singleton* non è un'operazione semplice. Inoltre *singleton* può essere problematico con il multi-threading.

Implementazione per sviluppo in realtà mista su Unity

Il pattern *singleton* può risultare molto utile nello sviluppo di applicazioni in realtà mista.

Un esempio tipico di utilizzo di questo pattern è per la gestione degli effetti sonori, in modo che ogni oggetto che riproduce un suono richiami il manager audio [45]. Se si decide di utilizzare gli oggetti d'interazione forniti da Mixed Reality ToolKit come spiegato al paragrafo 2.3, dei feedback sonori sono già integrati nell'oggetto. Può comunque essere utile un manager audio per questi oggetti se si vogliono utilizzare suoni specifici.

Un singleton in Unity è una classe accessibile globalmente che possiede un riferimento statico pubblico ad una singola istanza di quella stessa classe.

```
public class Singleton : MonoBehaviour
{
    public static Singleton instance;
}
```

Rendere statica una variabile significa che è condivisa da tutte le istanze della classe, il che significa che qualsiasi script può accedere al singleton tramite il nome della classe, senza bisogno del riferimento ad essa.

È possibile accedere al singleton in questo modo:

```
Singleton.instance;
```

In questo modo tutte le variabili e metodi del singleton sono facilmente accessibili da altri script dell'applicazione. Proprio per questo motivo generalmente è una buona idea utilizzare gli appositi get e set per consentire al-

le variabili di essere lette da qualsiasi script ma modificate solo nella classe singleton.

```
public class Singleton : MonoBehaviour
{
    public static Singleton Instance { get; private set; }
}
```

Il singleton deve sempre avere un'unica istanza e il singleton ha la responsabilità di assicurarsi che questo principio sia sempre valido. Per questo motivo quando viene creata o caricata l'istanza del singleton, si verifica se ci sono già altre istanze dell'oggetto attive. Con il metodo *Awake()* controlliamo se l'istanza creata è un duplicato, e nel caso si elimina da solo. *Awake()* è un metodo dell'interfaccia *MonoBehaviour* che viene chiamato al caricamento dell'istanza della classe.

```
public static Singleton Instance { get; private set; }
private void Awake()
{
    // If there is an instance, and it's not me, delete myself.

    if (Instance != null && Instance != this)
    {
        Destroy(this);
    }
    else
    {
        Instance = this;
    }
}
```

Finché la variabile o la funzione esiste sul singleton ed è accessibile pubblicamente, altri script saranno in grado di usarla senza che sia necessario impostare precedentemente un riferimento ad essa.

Riprendendo l'esempio iniziale, è possibile richiamare il singleton per riprodurre un effetto sonoro in un qualsiasi script del software in realtà mista in questo modo:

```
Singleton.Instance.PlaySound(clipToPlay);
```

3.3.5 Command

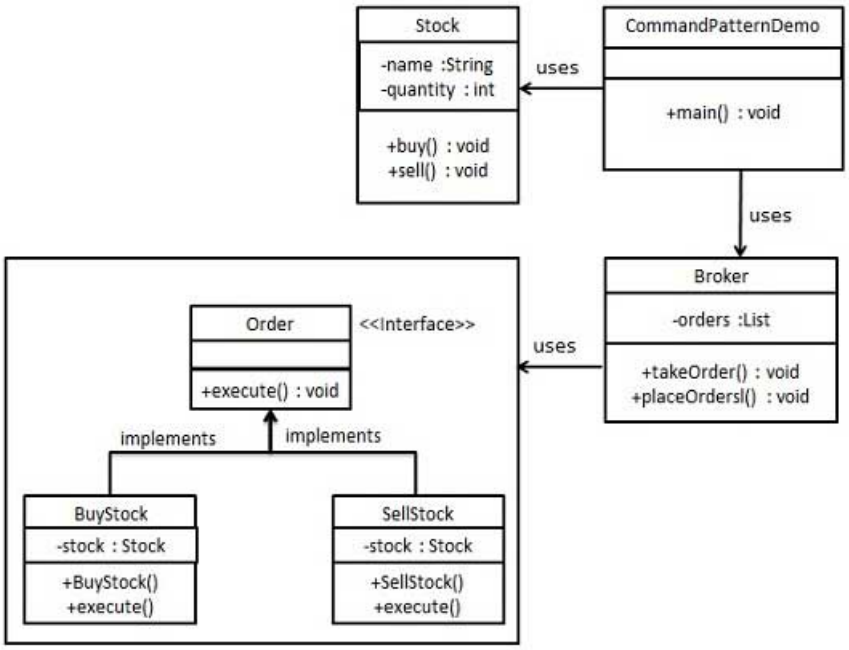


Figura 3.8: UML di un esempio di utilizzo di *Command* [46]

Descrizione del pattern

Command è un design pattern comportamentale, che ha come raggio d'azione gli oggetti.

Si tratta di uno dei pattern fondamentali e permette di isolare la parte di codice che esegue un'azione dal codice che ne richiede l'esecuzione.

Il pattern si basa su due elementi [47]:

- **Command**: che contiene il codice delle azioni che verranno eseguite.
- **Command Invoker**: che memorizza i comandi che dovranno essere eseguiti e li esegue al momento giusto, anche immediatamente se richiesto.

Vantaggi

Separando le azioni da eseguire dal codice che esegue l'esecuzione, risulterà più semplice la scrittura di codice e la sua lettura. Con questo modello di progettazione possiamo anche facilmente creare meccaniche di annullamento o ripetizione dei comandi.

Svantaggi

Con l'utlilzzo di *command* si dovrà lavorare con un maggior numero di classi e oggetti, per questo motivo il programmatore dovrà prestare attenzione a svilupparle correttamente[48].

Implementazione per sviluppo in realtà mista su Unity

Il pattern *Command* data la sua versatilità è applicabile ad innumerevoli contesti, anche a quello della realtà mista.

Sono molto presenti ad esempio ologrammi che hanno la sola funzione di controllo, MRTK ne fornisce una grande varietà e generalmente sono dei semplici pannelli che contengono diversi pulsanti (un esempio è visibile nella figura 2.7a). In questi casi risulta molto utile il pattern Command, che può essere implementato facilmente su Unity.

Saranno necessarie le classi "comando", che comprendono le implementazioni dei comandi stessi e la classe che invoca i comandi, che ha l'obbiettivo di immagazzinare i comandi ed eseguirli al momento giusto.

La classe comando avrà questa struttura:

```
// Abstract class for commands.
public abstract class Command
{
    // Method called to execute command.
    public abstract void Execute();
}
```

La classe che invoca i comandi avrà questa struttura:

```
// Collects command into buffer to execute them at once.
public class CommandInvoker : MonoBehaviour
{
    // Collected commands.
    private Queue<Command> commandBuffer
        = new Queue<Command>();

    public void AddCommand(Command command) { ... }

    public void ExecuteBuffer() { ... }
}
```

I due metodi fondamentali dell'invocatore di comandi sono:

- **AddCommand**: che aggiunge un comando alla coda.

```
// Method used to add new command to the buffer.
public void AddCommand(Command command)
{
    commandBuffer.Enqueue(command);
}
```

- **ExecuteBuffer**: che esegue tutti i comandi nella coda.

```
// Method used to execute all commands from the buffer.
public void ExecuteBuffer()
{
    foreach (var c in commandBuffer) { c.Execute(); }

    commandBuffer.Clear();
}
```

3.3.6 Model-View-Controller

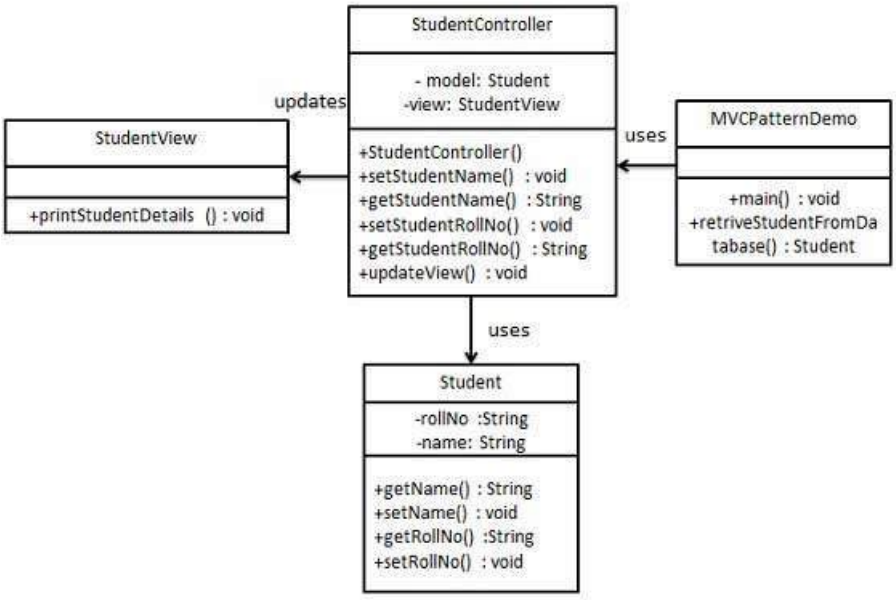


Figura 3.9: UML di un esempio di utilizzo di *Model-View-Controller* [49]

Descrizione del pattern

Model-View-Controller, noto come MVC, è un pattern architetturale. Perciò impatterà ad un livello più ampio lo sviluppo dell'applicazione, in particolare ne descriverà la sua struttura.

Con questo modello le classi e interfacce sono divise in tre sezioni, le quali hanno specifiche responsabilità:

- **View**: si occupa di gestire la parte di presentazione e di interazione con l'utente. Questa parte si focalizza sulla parte estetica dell'applicazione e consente di visualizzare nella maniera corretta i dati che riceve.
- **Model**: si occupa di gestire i dati e la logica dell'applicazione. Questa parte lavora direttamente con i dati, quindi le funzioni base saranno quelle di leggere, inserire, aggiornare ed eliminare gli stessi.
- **Controller**: gestisce la "meccanica" dell'applicazione, coordinando View e Model.

Le interazioni dell'utente nella View vengono catturate dal controller, il quale osserva, modifica o interroga il model, e quando necessario comanda modiche alla view. Non sono possibili altri tipi di interazione.

Vantaggi

Model-View-Controller rende lo sviluppo dell'applicazione più rapido, semplifica la scrittura di codice in team e semplifica anche l'aggiornamento e la manutenzione del software. Questa suddivisone netta della parte grafica dalla parte logica consente anche un cambio totale di interfaccia grafica senza complicazioni [50].

Svantaggi

MVC spesso non è un modello architetturale particolarmente semplice da comprendere, è quindi semplice commettere errori e violare i ruoli stringenti di ogni parte. Si tratta anche di un pattern piuttosto rigido, che deve rispettare regole specifiche. Gli svantaggi di Model-View-Controller non sono molti, e comunque di minore importanza rispetto ai numerosi benefici che comporta [50].

Implementazione per sviluppo in realtà mista su Unity

Il pattern architetturale *MVC* generalmente non è pensato per applicazioni in realtà mista, dal momento in cui gli engine più utilizzati forniscono già di base modelli architetturali diversi come il Component Pattern e il Game Loop Pattern. Nonostante questo è comunque possibile per una scelta prettamente di design del software optare per un modello di tipo Model-View-Controller su Unity.

Per creare il modello MVC è necessario implementare una classe controller. Spesso sono utilizzati più controller per le diverse aree dell'applicazione, in questo caso può essere una buona idea usare il pattern State Machine descritto al paragrafo 3.3.2 per cambiare facilmente lo stato del controller.

Per una facile gestione dei tipi di controller creiamo una enum in questo modo:

```
// SubControllers types.
public enum ControllerTypeEnum
{
    First,
    Second
}
```

Il controller principale avrà come campi i riferimenti ai sotto-controller ed i metodi per la loro gestione.

```
// Root controller responsible for changing phases
// with SubControllers.
public class RootController : MonoBehaviour
{
    // References to the subcontrollers.
    private FirstController firstController;
    private SecondController secondController;

    private void Start() { ... }

    public void ChangeController(ControllerTypeEnum c) { ... }

    public void DisengageControllers() { ... }
}
```

I metodi del controller principale sono:

1. **Start**: chiamato all'avvio del controller e imposta il primo sotto-controller.

```
// Unity method called on first frame.
private void Start()
{
    firstController.root = this;
    secondController.root = this;
    ChangeController(firstController);
}
```

2. **ChangeController**: esegue la transizione da un sotto-controller ad un altro.

```
// Method used by subcontrollers to change phase.
public void ChangeController(ControllerTypeEnum c)
{
    // Reseting subcontrollers.
    DisengageControllers();

    // Enabling subcontroller based on type.
    switch (c)
    {
        case ControllerTypeEnum.First:
            firstController.EngageController();
            break;
        case ControllerTypeEnum.Second:
            SecondController.EngageController();
    }
}
```

3. **DisengageControllers**: disabilita il sotto-controller attuale.

```
public void DisengageControllers()
{
    firstController.DisengageController();
    secondController.DisengageController();
}
```

La classe degli specifici sotto-controller ha come campo il riferimento al controller principale e due metodi che lo attivano o lo disattivano:

```
// Base class for SubControllers with reference
// to Root Controller.
public abstract class SubController : MonoBehaviour
{
    public RootController root;

    // Method used to engage controller.
    public virtual void EngageController()
    { gameObject.SetActive(true); }

    // Method used to disengage controller.
    public virtual void DisengageController()
    { gameObject.SetActive(false); }
}
```

È possibile estendere la classe con una generica, per permettere ad ogni sotto-controller di avere la propria view specifica:

```
// Extending SubController class with generic reference UI.
public abstract class SubController<T> : SubController
    where T : UIRoot
{
    protected T ui;
    public T UI => ui;

    public override void EngageController()
    {
        base.EngageController();
        ui.ShowRoot();
    }

    public override void DisengageController()
    {
        base.DisengageController();
        ui.HideRoot();
    }
}
```

Con un sotto-controller generico, implementiamo la classe view in questo modo:

```
// Base class for UI roots for different controllers.
public class UIRoot : MonoBehaviour
{
    // Method used to show UI.
    public virtual void ShowRoot()
    {
        gameObject.SetActive(true);
    }

    // Method used to hide UI.
    public virtual void HideRoot()
    {
        gameObject.SetActive(false);
    }
}
```

Per la parte di model, che quindi si occupa della gestione delle informazioni che gestisce il software, può essere conveniente usare il pattern Singleton descritto al paragrafo 3.3.4, così da salvare ed accedere ai dati comodamente.

Conclusioni

Dal lavoro di tesi è emerso che gli attuali framework di sviluppo, in particolare Unity di cui è stata svolta un'analisi più dettagliata, integrati con specifici strumenti come MRTK risultino la scelta migliore per programmare un software in realtà mista.

Grazie a Mixed Reality ToolKit infatti sono disponibili numerosi strumenti utili allo sviluppatore senza bisogno di crearli da zero, inoltre l'integrazione con l'API OpenXR facilita la portabilità tra diversi dispositivi. Dalle analisi effettuate è stata anche delineata una discreta flessibilità per quanto riguarda la libertà dello sviluppatore di scegliere di implementare i principali design pattern, infatti tutti i pattern presi in esame sono risultati realizzabili su Unity senza particolari problemi.

Nonostante l'utilizzo di Unity e Unreal Engine ad oggi siano nella maggior parte dei casi la scelta migliore rispetto allo sviluppo in codice nativo, si percepisce che i framework non sono stati creati a monte per lo sviluppo in realtà aumentata, virtuale e mista. Sarebbe importante quindi che in futuro venga creato un ambiente di sviluppo specifico per la realtà estesa, con già il supporto a tutte le tecnologie utili come OpenXR, e man mano che i progetti aumentano di complessità conseguentemente crescerà la necessità di questo nuovo ambiente.

La tesi ha anche evidenziato quali sono i pattern più utili nella realizzazione di un software in realtà mista, dai design pattern più comuni come il Game Loop Pattern, ad altri nativi per la realtà aumentata e mista, come Scene Graph Pattern o Scripting Pattern. Essendo queste tecnologie recenti e in forte sviluppo porteranno alla nascita di nuovi pattern specifici per le singole tecnologie della realtà estesa.

Ringraziamenti

Innanzitutto, vorrei ringraziare il Professor Alessandro Ricci e l'Ingegner Samuele Burattini, rispettivamente relatore e correlatore della tesi, per avermi seguito durante la stesura della tesi con professionalità e puntualità.

Un ringraziamento speciale va alla mia famiglia, che mi ha sempre sostenuto e creduto in me.

Vorrei ringraziare anche i miei colleghi universitari e i miei compagni di squadra, in particolare Marco e Alice, che mi hanno accompagnato in questo percorso rendendolo uno dei periodi più belli della mia vita.

Infine, ringrazio Celine, per essere sempre stata al mio fianco.

Bibliografia

[1] Reality–virtuality continuum.
https://medium.com/desn325-emergentdesign/medium-post-1-reality-virtuality-continuum-1c2f490de157.

[2] Collection: Curatorial Strategies and Post digital Practices.
https://anti-materia.org/academic-resources.

[3] What is Google Maps AR navigation, Live View, and how do you use it?
https://www.pocket-lint.com/apps/news/google/147956-what-is-google-maps-ar-navigation-and-how-do-you-use-it.

[4] Billionaires See VR as a Way to Avoid Radical Social Change.
https://www.wired.com/story/billionaires-use-vr-avoid-social-change/.

[5] Utilization of Geographic Data for the Creation of Occlusion Models in the Context of Mixed Reality Applications.
https://link.springer.com/chapter/10.1007/978-3-031-15553-6_18.

[6] Learn about HoloLens 2 features and review technical specs.
https://www.microsoft.com/en-us/hololens/hardware.

[7] Sean Ong. Beginning windows mixed reality programming.
Berkeley, CA: Apress. doi, 10:978–1, 2017.

[8] Understand display techniques in augmented reality.
https://niteeshyadav.com/blog/understanding-display-techniques-in-augmented-reality-7485/.

[9] The most immersive enterprise AR device.
https://www.magicleap.com/en-us/.

[10] Magic Leap 1.
https://www.boxxe.com/magic-leap.

[11] Head Up Display on Mercedes-Benz Vehicles.
https://www.raycatenaunion.com/head-up-display-mercedes-benz/.

[12] Are Wearables the Next Big Thing for AR?
https://www.iotforall.com/future-of-augmented-reality-enabled-wearable-devices.

[13] Start designing and prototyping.
https://learn.microsoft.com/en-us/windows/mixed-reality/design/design.

[14] What is a hologram?
https://learn.microsoft.com/en-us/windows/mixed-reality/discover/hologram.

[15] Holographic frame.
https://learn.microsoft.com/en-us/windows/mixed-reality/design/holographic-frame.

[16] Design Magic Leap 1 Control.
https://ml1-developer.magicleap.com/en-us/learn/guides/design-magic-leap-one-control.

[17] Introducing instinctual interactions.
https://learn.microsoft.com/en-us/windows/mixed-reality/design/interaction-fundamentals.

[18] Hands and motion controllers.
https://learn.microsoft.com/it-it/windows/mixed-reality/design/hands-and-tools.

[19] Gaze and commit.
https://learn.microsoft.com/en-us/windows/mixed-reality/design/gaze-and-commit.

[20] Hands-free.
https://learn.microsoft.com/en-us/windows/mixed-reality/design/hands-free.

[21] Spatial mapping.
https://learn.microsoft.com/en-us/windows/mixed-reality/design/spatial-mapping.

[22] Shared experiences in mixed reality.
https://docs.microsoft.com/en-us/windows/mixed-reality/design/shared-experiences-in-mixed-reality#get-started-building-shared-experiences.

[23] Getting started with the HoloLens 2 and Unity.
https://learn.microsoft.com/en-us/shows/mixed-reality/getting-started-with-the-hololens-2-and-unity.

[24] Unity Learn: Learn game development with Unity.
https://learn.unity.com/.

[25] Unity tutorials point.
https://www.tutorialspoint.com/unity/unity_tutorial.pdf.

[26] Andrew Sanders. *An introduction to Unreal engine 4.*
AK Peters/CRC Press, 2016.

[27] OpenXR Development Services Hire OpenXR Developer.
https://www.softwebsolutions.com/openxr-development-services.html.

[28] Microsoft aderisce a OpenXR lo standard aperto Royalty-free per VR e AR.
https://www.vr-italia.org/microsoft-aderisce-a-openxr-lo-standard-aperto-royalty-free-per-vr-e-ar/.

[29] What is Mixed Reality Toolkit 2?
https://learn.microsoft.com/en-us/windows/mixed-reality/mrtk-unity/mrtk2/?view=mrtkunity-2022-05.

[30] James William Cooper. Java design patterns: a tutorial. 2000.

[31] Cosa sono i Design Pattern perché si usano o perché si dovrebbero usare.
https://www.dgroove.it/cosa-sono-i-design-pattern-perche-si-usano-o-perche-si-dovrebbero-usare/4099/.

[32] Asa MacWilliams, Thomas Reicher, Gudrun Klinker, and Bernd Bruegge. Design patterns for augmented reality systems. In *Proceedings of the International Workshop exploring the Design and Engineering of Mixed Reality Systems (MIXER), Funchal, Madeira, CEUR Workshop Proceedings*, 2004.

[33] Scene Graphs (Advanced Methods in Computer Graphics).
http://what-when-how.com/advanced-methods-in-computer-graphics/scene-graphs-advanced-methods-in-computer-graphics-part-1/.

[34] Unity Architecture Pattern: the Main loop.
https://bronsonzgeb.com/index.php/2021/04/24/unity-architecture-pattern-the-main-loop/.

[35] Writing An SDL2 Game Loop.
http://matthewstyles.com/writing-an-sdl2-game-loop/.

[36] Robert Nystrom. *Game programming patterns*. Genever Benning, 2014.

[37] Game Loop.
https://gameprogrammingpatterns.com/game-loop.html.

[38] Order of execution for event functions.
https://docs.unity3d.com/manual/executionorder.html.

[39] Design Patterns State Pattern.
https://www.tutorialspoint.com/design_pattern/state_pattern.htm.

[40] How to implement State Machine in Unity.
https://www.patrykgalach.com/2019/03/18/design-pattern-state-machine/.

[41] State Design Pattern.
https://deviq.com/design-patterns/state-design-pattern.

[42] Design Pattern Factory Pattern.
https://www.tutorialspoint.com/design_pattern/factory_pattern.htm.

[43] Design Pattern Advantages and Disadvantages of Factory Design Pattern.
https://www.buggybread.com/2016/12/design-pattern-advantages-and.html.

[44] Design Pattern Singleton Pattern.
https://www.tutorialspoint.com/design_pattern/singleton_pattern.htm.

[45] Singletons in Unity (done right).
https://gamedevbeginner.com/singletons-in-unity-the-right-way/.

[46] Design Patterns Command Pattern.
https://www.tutorialspoint.com/design_pattern/command_pattern.htm.

[47] Implementing Command Design Pattern in Unity.
https://www.patrykgalach.com/2019/06/03/command-design-pattern-in-unity/.

[48] Advantages and disadvantages of Command patterns. https://www.oreilly.com/library/view/learning-python-design/9781785888038/ch07s04.html.

[49] Design Patterns MVC Pattern. https://www.tutorialspoint.com/design_pattern/mvc_pattern.htm.

[50] What is MVC? Advantages and Disadvantages of MVC. https://www.interserver.net/tips/kb/mvc-advantages-disadvantages-mvc/.