Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea Magistrale in Informatica

Sviluppo di una libreria Android per la creazione di esperienze in Realtà Aumentata legate a Punti di Interesse

Relatore:
Chiar.mo Prof.
Luciano Bononi

Presentata da:
Davide Marchi

Correlatore:
Dott.
Luca Bedogni

Sessione II
Anno Accademico 2017-2018

A Rini, la mia piccola doghina grassa.

Introduzione

Nel 2018 ci sono circa 4 miliardi di utenti connessi ad Internet [1], circa il 55% della popolazione mondiale, di cui circa 3 miliardi possiede uno smartphone [2]. Le persone utilizzano i propri device per pianificare la loro giornata, controllare i propri impegni oppure per cercare informazioni utili, tutto questo grazie anche ad un gran numero di applicazioni capaci di soddisfare qualsiasi necessità. Possiamo quindi affermare che, dall'inizio del nuovo millennio, grazie al rapido sviluppo di nuove tecnologie i metodi di lavoro, di collaborazione e di interazione tra le persone sono cambiati. Proprio per questo motivo è nata anche una forte esigenza di sviluppare applicazioni vincenti dal punto di vista comunicativo, che siano semplici nell'utilizzo ma complete di funzionalità con lo scopo di essere comprensibili da chiunque in maniera immediata. Questa necessità ha portato alla crescita di nuove metodologie di sviluppo improntate molto sulla *User Experience* (*UX*) tramite le quali poter realizzare soluzioni comprensibili in grado di comunicare in modo ottimale con gli utilizzatori finali.

Uno dei motivi per cui nasce questa tesi è proprio quello di trovare una soluzione per migliorare la *UX* degli utenti nell'utilizzo dei propri device, in particolare lavoreremo su due fattori chiave: i *punti di interesse*, intesi come luoghi fisici nel mondo, e la loro persistenza nel *tempo*. Per poter proporre una possibile soluzione, analizzeremo quali sono le tecnologie presenti sul mercato che si propongono di risolvere questa problematica generale e cercheremo di capire quali sono i vantaggi e gli svantaggi di ogni soluzione. Per motivazioni che giustificheremo nel corso dei prossimi capitoli, tra

le tecnologie che andremo ad analizzare metteremo al centro della soluzione che implementeremo la *Realtà Aumentata*: tecnologia sempre più popolare e diffusa nella vita quotidiana delle persone grazie alla capacità di arricchire la percezione sensoriale umana tramite la sovrapposizione di informazioni e contenuti virtuali al mondo reale, capacità che sfrutteremo per cercare di migliorare l'esperienza degli utenti. Una volta terminate queste analisi, come prodotto principale della tesi andremo a sviluppare una libreria Android che sfrutti proprio la Realtà Aumentata per la creazione di esperienze legate a punti di interesse.

I prossimi capitoli saranno quindi articolati nel seguente modo: inizialmente andremo ad esporre tutte le analisi sulla Realtà Aumentata e sul problema preso in esame, successivamente passeremo alla progettazione ed allo sviluppo della soluzione proposta ed infine mostreremo un prototipo che implementerà proprio questa soluzione, grazie al quale dimostreremo quali sono le sue funzionalità e potenzialità.

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Descrizione del contesto

1.1 Realtà Aumentata

La Realtà Aumentata (AR) può essere definita come la visualizzazione in tempo reale del mondo che ci circonda "aumentata" da informazioni ed oggetti virtuali computer-generated. A differenza della Realtà Virtuale (VR) in cui l'utente è completamente immerso in un mondo virtuale, nell'AR possiamo quindi vedere il mondo reale attorno a noi arricchendo la percezione sensoriale umana mediante informazioni, in genere manipolate e convogliate elettronicamente, che non sarebbero percepibili con i cinque sensi [3].

La VR è immersiva ed isola l'utente dall'ambiente esterno trasportandolo in un mondo virtuale creato con leggi fisiche e proprietà che non sempre rispecchiano quelle del mondo reale. Per far sì che l'utente sia immerso in questo mondo sintetico, la VR fa uso di periferiche quali visori, auricolari, guanti o tute che avvolgono interamente il corpo. Nell'AR invece i contenuti virtuali coesistono con gli oggetti reali, infatti Ronald Azuma in alcune sue pubblicazioni [4] [5] definisce l'AR come un qualunque sistema che abbia le seguenti 3 principali caratteristiche:

1. capacità di combinare oggetti reali e virtuali in un ambiente reale;

2. interazione in real-time con il mondo reale;

3. capacità di considerare tutte e 3 le dimensioni, ovvero non è un semplice overlay 2D di informazioni sopra il mondo reale.

Non dobbiamo quindi considerare l'AR come un semplice rendering di immagini bidimensionali o tridimensionali attraverso un device, ma piuttosto come uno strato virtuale di informazioni che viene posizionato sopra il mondo reale per come lo conosciamo. La Realtà Aumentata inoltre non è limitata solamente alla vista ma possiamo applicarla a tutti i sensi, potendola addirittura utilizzare per sostituire dei sensi mancanti [6].

In una pubblicazione del 1994 Paul Milgram posiziona la VR e l'AR all'interno di ciò che definisce *Reality-Virtuality Continuum*: a sinistra troviamo ciò che possiamo osservare guardando il mondo reale (*Real Environment*), a destra troviamo un mondo sintetico composto esclusivamente da oggetti virtuali (*Virtual Environment*), mentre tra questi due estremi troviamo tutte le possibili variazioni tra reale e virtuale [7].

La parte centrale viene definita come Mixed Reality (MR) e rappresenta tutti gli ambienti in cui i contenuti virtuali e quelli reali sono mostrati insieme in una singola visualizzazione (Figura 1.1).

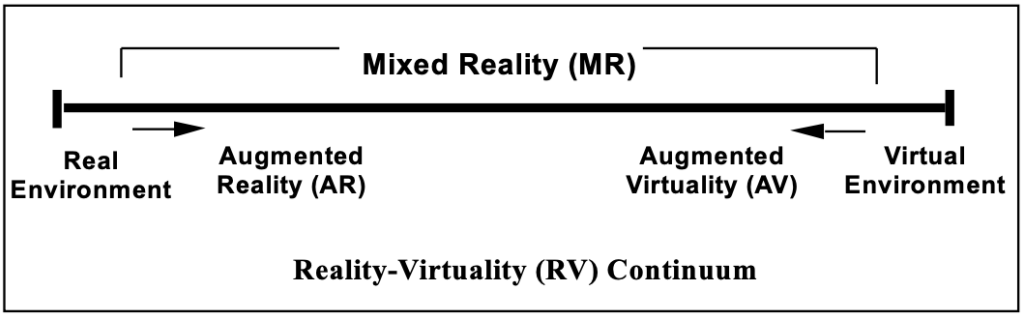

Figura 1.1: Rappresentazione del Reality-Virtuality Continuum [7].

A differenza dell'AR in cui gli oggetti virtuali sono mostrati in primo piano rispetto al mondo reale, nella MR sono perfettamente integrati e potenzialmente possono interagire con esso (Figura 1.2).

Figura 1.2: Da sinistra verso destra: Virtual Reality, Augmented Reality e Mixed Reality [8].

Nell'ultimo decennio nasce anche il concetto di *Mobile Augmented Reality*, ovvero l'applicazione della Realtà Aumentata allo sviluppo di applicazioni per dispositivi mobile o wearable.

Tutti i concetti esposti finora fanno parte dell'insieme che oggi viene chiamato *Extended Reality* (XR), termine in continua evoluzione utilizzato per denotare l'estensione delle esperienze umane relative ai sensi tramite contenuti virtuali [9].

1.2 Cenni storici

Nel lontano 1901 L. Frank Baum, uno scrittore, menzionò per la prima volta l'idea di un display elettronico che permettesse di sovrapporre delle informazioni sopra il mondo reale, nello specifico sopra le persone, e gli diede il nome di *Character Marker* [10]. Grazie a questo dispositivo, chiunque lo indossasse sarebbe stato in grado di vedere sopra alle persone che incontrava un *marker* con una lettera che indicava qual era il carattere di quella persona, in modo tale da capire le sue vere intenzioni e la sua vera natura. Ad esempio le persone buone sarebbero state indicate con la G (*Good*) e quelle malvagie con la E (*Evil*), le persone sagge con la W (*Wise*) mentre i folli con la F (*Foolish*), e così via.

Successivamente, tra la fine degli anni '50 e l'inizio degli anni '60, il

cinematografo Morton Heiling ipotizzò *"The Cinema of the Future"*, ovvero un modo per coinvolgere tutti i sensi dello spettatore in ciò che stava vedendo. Nel 1962 creò e brevettò[1] il suo simulatore chiamato *Sensorama*, il quale può essere considerato come uno dei primi esempi di tecnologia immersiva e multi-sensoriale (Figura 1.3). La macchina era un dispositivo meccanico che includeva un display a colori stereoscopico, dei ventilatori per simulare il vento, degli emettitori di odori, un sistema audio ed una sedia che si poteva muovere in varie direzioni.

Figura 1.3: *Sensorama* di Morton Heiling, 1962 [11].

La prima apparizione della Realtà Virtuale risale al 1968, anno in cui Ivan Sutherland, informatico e professore di Harvard, con l'aiuto di alcuni suoi studenti creò il primo *Head-Mounted Display* (HMD), ovvero un dispositivo indossabile composto da un display che pendeva dal soffitto chiamato *The*

[1]Brevetto Sensorama, `http://www.mortonheilig.com/SensoramaPatent.pdf`.

Sword of Damocles [12]. Questo dispositivo viene considerato tuttora come il primo sistema di Realtà Virtuale su tecnologia indossabile (Figura 1.4).

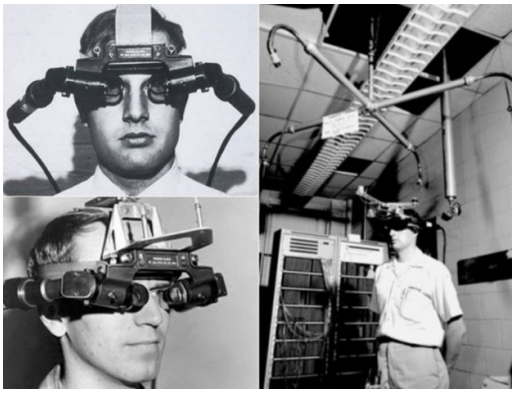

Figura 1.4: *The Sword of Damocles* di Ivan Sutherland, 1968 [13].

L'interesse per questa nuova dimensione portò nel decennio tra il 1970 ed il 1980 ad uno studio approfondito sull'argomento e gli sviluppi fecero un enorme salto nel 1975 grazie al progetto *Videoplace* di Myron Krueger, un laboratorio di *realtà artificiale* in cui venivano combinati un sistema di proiezione e delle videocamere al fine di produrre ombre sullo schermo. Come è possibile vedere in Figura 1.5, proiettando le silhouettes degli utenti *Videoplace* permetteva un loro coinvolgimento in un ambiente interattivo.

Figura 1.5: *Videoplace* di Myron Krueger, 1975 [14].

Cinque anni più tardi la ricerca di Gavan Lintern dell'Università dell'Illinois fu il primo lavoro pubblicato che mostrò le potenzialità dell'utilizzo di un HMD per insegnare ai piloti come far atterrare un aereo, dimostrando che i soggetti che si erano allenati nell'atterraggio utilizzando questo simulatore ottenevano risultati migliori rispetto ai soggetti che non ne avevano fatto uso [15]. Lo stesso anno Steve Mann creò la prima versione di quello che successivamente chiamerà *EyeTap*, ovvero il primo wearable computer dotato di un sistema che permetteva la sovrapposizione di informazioni testuali e grafiche sulla scena reale catturata (Figura 1.6).

Figura 1.6: Steve Mann ed il suo primo wearable computer, 1980 [16].

Nel 1981 fu Dan Reitan a portare un concetto precursore della Realtà Aumentata nelle trasmissioni televisive utilizzando delle mappe meteorologiche geospazializzate per riprodurre le immagini meteorologiche prese dai radar sopra a quelle terrestri.

Tre anni dopo uscì nelle sale il primo film della saga *The Terminator* in cui il protagonista utilizza la tecnologia all'interno del proprio corpo per scansionare oggetti e persone al fine di ottenere informazioni testuali utili che vengono visualizzate sovrapposte al mondo reale (Figura 1.7).

Un ulteriore concetto precursore della Realtà Aumentata lo troviamo nel 1987 grazie alla creazione di un prototipo funzionante di un *head-up display*

Figura 1.7: Scena dal film *The Terminator* di James Cameron, 1984 [17].

(HUD) basato su un telescopio astronomico, il quale permetteva di mostrare immagini di stelle e costellazioni direttamente nella lente del telescopio, sovrapponendole e sincronizzandole alle immagini reali del cielo [18].

Negli anni '90 sono nate le prime visioni coerenti ed organizzate di come l'elettronica miniaturizzata, i dispositivi portatili, Internet e la geolocalizzazione potessero essere utilizzati per creare mondi virtuali o per arricchire il mondo reale. É proprio nel 1990 che Thomas P. Caudell conia il termine *Augmented Reality* per descrivere la sovrapposizione di elementi virtuali a scene reali. Questo avviene quando lui e David Mizell, entrambi ricercatori Boeing, vengono incaricati di trovare una soluzione alle difficoltà incontrate dagli elettricisti aeronautici nel cablaggio di alcune parti all'interno degli aerei. A tal scopo, realizzarono un display digitale in grado di visualizzare l'esatta sequenza di operazioni necessarie per apportare gli interventi di assemblaggio in modo corretto, evitando agli operai la dispendiosa e difficoltosa operazione di dover consultare manuali e tavole di compensato con le istruzioni di cablaggio progettate individualmente per ogni parte dell'aereo mentre si trovavano in spazi stretti ed angusti [19].

Il primo sistema funzionante di AR risale però al 1992 e fu sviluppato da Louis Rosenberg presso lo U.S. Air Force Armstrong's Research Laboratory. Il progetto si chiamava *Virtual Fixtures* ed era un complesso sistema robotico

che permetteva una sovrapposizione di informazioni sensoriali al fine di migliorare la normale percezione dell'ambiente reale da parte dell'utilizzatore, migliorandone la produttività e le capacità sia per lavori sul posto che per lavori manipolati da remoto (Figura 1.8) [20].

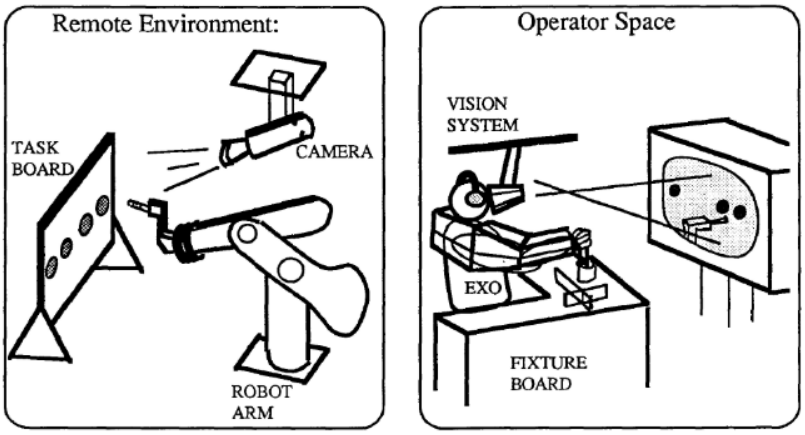

Figura 1.8: Setup del sistema *Virtual Fixtures* di Louis Rosenberg, 1992 [20].

Solo un anno più tardi iniziano ad uscire vari lavori sull'AR, come ad esempio il progetto *KARMA* (Knowledge-based Augmented Reality for Maintenance Assistance) presentato da Steven Feiner, Blair MacIntyre e Doree Seligmann, il quale era il prototipo di un sistema che sfruttava l'AR per mostrare semplici operazioni di manutenzione su stampanti laser tramite un HMD (Figura 1.9) [21], oppure il progetto di Mike Abernathy in cui l'AR veniva utilizzata per l'identificazione di detriti spaziali sovrapponendo le traiettorie geografiche satellitari sul video live di un telescopio [22].

Nel 1994 Paul Milgram e Fumio Kishino introdussero il concetto di *Mixed Reality* [7], mentre contemporaneamente la Realtà Aumentata fece il suo debutto all'interno dei teatri: Julie Martin creò *Dancing in Cyberspace*, la prima produzione teatrale in Realtà Aumentata dove degli acrobati danzavano insieme ed attorno ad oggetti virtuali proiettati sul palco.

Due anni più tardi venne creato *CyberCode*[2], il primo sistema in Realtà Aumentata ad utilizzare dei marker 2D. *CyberCode* diventerà successiva-

[2]CyberCode Demo 1996, `https://www.youtube.com/watch?v=cNFwEMkLK4A`.

Figura 1.9: Progetto *KARMA*, 1993 [21].

mente il modello di partenza per lo sviluppo di sistemi AR marker-based [23].

Nel periodo dal 1997 al 2001 la Canon Inc. ed il governo Giapponese finanziarono il *Mixed Reality Systems Laboratory* per fare ricerche sulla Realtà Aumentata e sulla Realtà Virtuale [24], mentre la Columbia University sviluppò il primo sistema outdoor in AR chiamato *The Touring Machine* [25], introducendo anche la definizione di *Mobile Augmented Reality System* (MARS) [26]. Sempre nel 1997 Ronald Azuma scrisse il primo saggio sulla Realtà Aumentata [4].

L'AR iniziò ormai a farsi spazio anche nell'ambito dell'intrattenimento delle persone in qualità di tecnologia avanzata, infatti nel Settembre del 1998 venne trasmessa in onda da Sportvision la prima versione della *virtual yellow first-down line* in una partita della National Football League sfruttando un sistema chiamato *1st & Ten*. L'elemento grafico posizionato virtualmente sul campo di gioco è tuttora utilizzato per mostrare ai telespettatori dov'è posizionata la linea del primo down (Figura 1.10). Sempre nel 1998, presso l'Università del North Carolina, Ramesh Raskar introduce la *Spatially Augmented Reality*, tecnologia in grado di proiettare oggetti virtuali direttamente

Figura 1.10: Esempio di *yellow first-down line* nel football americano.

nello spazio fisico dell'utente senza la necessità di indossare un HMD [27].

Un anno più tardi i ricercatori dello U.S. Naval Research Laboratory (NRL) iniziarono a lavorare sul progetto *Battlefield Augmented Reality System* (BARS) [28], il quale durerà decenni, per prototipizzare un dispositivo wearable per formare ed addestrare i soldati ad operare in ambienti urbani. Contemporaneamente Frank Delgado e Mike Abernathy realizzarono un *Hybrid Synthetic Vision System* che sfrutta l'AR per sovrapporre informazioni alle mappe in modo tale da fornire una navigazione visiva migliorata durante i test di volo. Inoltre fecero volare con successo il veicolo X-38 della NASA dotato di questo sistema [29] [30].

L'interesse per la Realtà Aumentata continuò a crescere, diventando fonte di studio anche da parte di molti amatori: è proprio in questo contesto che vennero creati tools e frameworks a supporto dello sviluppo di applicazioni per sfruttare l'AR. Tra questi tools il più famoso è sicuramente quello sviluppato da Hirokazu Kato nel 1999 presso lo Human Interface Technology Laboratory (HITLab): *ARToolKit*[3], una libreria open-source per la creazione di applicazioni in Realtà Aumentata. Sempre nel 1999 Steve Mann, il padre del wearable computing, creò ufficialmente *EyeTap* [31].

[3]ARToolKit, `https://github.com/artoolkit`.

Nel 2000 uscì *ARQuake* [32], il primo gioco mobile outdoor in AR sviluppato da Bruce H. Thomas ispirato allo spara-tutto in prima persona *Quake* uscito nel 1996. In Figura 1.11 possiamo vedere una schermata del gioco e l'equipaggiamento necessario per poter ottenere una posizione GPS accurata dell'utente ed i valori corretti sulla sua orientazione.

Figura 1.11: A sinistra una schermata del gioco *ARQuake*, a destra l'equipaggiamento da indossare per poterci giocare, 2000 [32].

Un anno più tardi Reitmayr e Schmalstieg presentarono un sistema mobile in Realtà Aumentata che permetteva a più utenti di lavorare contemporaneamente in uno spazio di lavoro condiviso ed aumentato [33], mentre Vlahakis *et al.* presentarono *Archeoguide*, un sistema mobile in AR per visitare i siti archeologici di Olimpia, in Grecia [34].

Per la stagione NFL 2003, Sportvision rivelò il primo sistema di computer graphics in grado di inserire la *1st & Ten line* tramite la *Skycam*: la telecamera che fornisce una prospettiva aerea del campo da gioco.

L'anno successivo la Trimble Navigation e lo HITLab dimostrarono un sistema outdoor in AR montato su un casco che fornisce informazioni sull'ambiente circostante (Figura 1.12) [35].

Attorno al 2005 furono sviluppati sistemi per fotocamere in grado di analizzare il mondo fisico in tempo reale, ottenendo la posizione degli oggetti e

Figura 1.12: Contenuti visibili tramite il casco realizzato dalla Trimble Navigation e lo HITLab, 2004 [35].

le relazioni fra loro. Grazie a questo passo avanti, i sistemi basati su fotocamera diventarono la base per la maggior parte delle tecnologie di supporto alla Realtà Aumentata [6].

Negli anni seguenti l'AR si diffuse anche nell'ambito mobile grazie ad alcune librerie innovative come *Wikitude* e *Metaio*, le quali fornivano agli sviluppatori delle API che rendevano meno complicato l'approccio alla Realtà Aumentata. Proprio per dimostrare le potenzialità di queste APIs, nell'Ottobre 2008 Wikitude rilasciò la *Wikitude AR Travel Guide*, un'applicazione mobile che sfruttava la Realtà Aumentata per mostrare un layer di informazioni testuali relative al luogo che si stava inquadrando con la fotocamera del device. Contemporaneamente l'AR approdò anche nel mondo del marketing grazie alla campagna pubblicitaria realizzata dalla BMW che incoraggiava i consumatori ad inquadrare il magazine con le proprie webcam per visualizzare un modello 3D della Mini Cabrio appena lanciata sul mercato (Figura 1.13) [36].

Nel 2009, grazie all'integrazione di *ARToolKit* in Adobe Flash (FLARToolKit[4]) realizzata da Saqoosha, la Realtà Aumentata entrò a fare parte

[4]FLARToolKit, `http://www.libspark.org/wiki/saqoosha/FLARToolKit/en`.

Figura 1.13: Modello di BMW Mini Cabrio visualizzato in AR, 2008 [37].

dei browser web, mentre *Esquire* inserì delle pagine di un loro magazine contenuti in Realtà Aumentata che sarebbero apparsi inquadrando la pagina con una fotocamera [38].

Tre anni dopo vennero lanciati sul mercato *Lyteshot*[5], una piattaforma interattiva di gioco in AR che utilizza un paio di smart glasses per le informazioni relative al gioco, il *Meta 1 Developer Kit* con il relativo headset [39] e diventarono disponibili i *Google Glass* (Figura 1.14) per una cerchia ristretta di sviluppatori selezionati come tester. Poco dopo le case automobilistiche utilizzarono la Realtà Aumentata per sostituire i manuali delle informazioni: ad esempio Volkswagen ed Audi lanciarono sul mercato, rispettivamente, le applicazioni mobile *MARTA* (Mobile Augmented Reality Technical Assistance) [40] ed Audi AR [41].

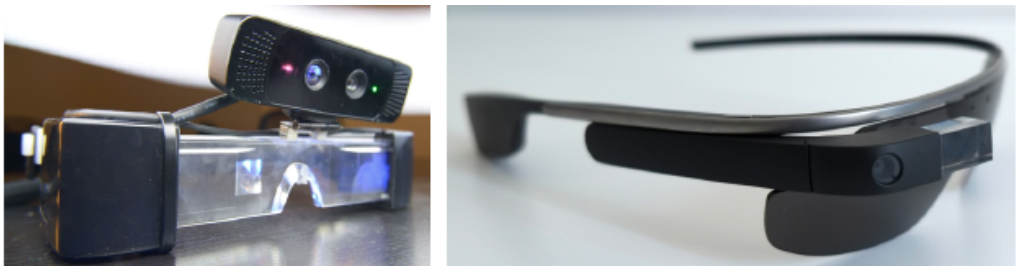

Figura 1.14: A sinistra il *Meta 1* [39], a destra i *Google Glass* [42], 2012.

[5]LyteShot, `http://www.lyteshot.com`.

Nel 2014 i *Google Glass* vennero rilasciati ufficialmente al pubblico, ma pur non avendo il successo sperato dagli sviluppatori fin da subito, dimostrarono il vero potenziale della Realtà Aumentata applicata al wearable computing. Proprio per questo motivo, poco dopo *BlippAR* sviluppò la prima piattaforma[6] per lo sviluppo di giochi in AR per *Google Glass* [43]. In febbraio Google annunciò il *Project Tango*, che consisteva nel realizzare, in collaborazione con alcuni produttori scelti, una serie di smartphone dotati di sensori speciali per portare il mondo della Realtà Aumentata tra le mani di tutti [44]. Inoltre il 2014 fu l'anno in cui uscì la prima generazione di giochi che sfruttarono l'AR a scopo educativo, grazie allo sviluppo di *i-Wow Atlas World*[7] da parte di *Mahei*, ed in cui *Magic Leap* ricevette 542 milioni di dollari: il più grande finanziamento ricevuto fino ad ora nel campo della Realtà Aumentata [45] [46].

L'anno successivo Microsoft annunciò la piattaforma di Realtà Aumentata *Windows Holographic* ed il *Microsoft HoloLens*[8], il visore senza cavi progettato appositamente per sfruttare le potenzialità della piattaforma. *HoloLens* sfrutta un display ottico 3D, un sistema di scansione spaziale dei suoni e vari tipi di sensori avanzati, il tutto unito ad una *Holographic Processing Unit* che integra i vari dati, per poter fornire un'interfaccia con cui interagire tramite lo sguardo, i comandi vocali o i gesti delle mani.

Il 2016 fu un anno di svolta: gli investimenti nel campo della AR raggiunsero il miliardo di dollari [47] e con il rilascio di *Pokémon Go* per iOS ed Android da parte della *Niantic, Inc.*, gioco che diventò immediatamente una delle applicazioni mobile più scaricate ed utilizzate con un picco di 46 milioni di utenti giornalieri, la popolarità della Realtà Aumentata incrementò notevolmente (Figura 1.15). Inoltre nel corso di quest'anno vennero anche lanciati sul mercato la nuova versione del *Microsoft HoloLens* ed il *Meta 2 Developer Kit*.

[6] Piattaforma BlippAR, `https://www.youtube.com/watch?v=ZBCBXWSx-EE`.

[7] i-Wow Atlas World, `http://www.mahei.es/iwowatlas.php?lang=en`.

[8] HoloLens, `https://www.microsoft.com/en-us/hololens`.

Figura 1.15: Schermata di gioco di *Pokémon Go*, 2016.

Nel 2017 *Magic Leap* annunciò il sistema *Magic Leap One headset*[9] per la Mixed Reality, il quale sarà disponibile dal 2018 insieme a tutto l'occorrente per gli sviluppatori (SDK, tool e documentazione). Durante il corso dell'anno vennero anche annunciate *ARKit*[10] da parte di Apple e *ARCore*[11] da parte di Google, le quali resero e renderanno lo sviluppo di applicazioni in AR molto più semplice ed accessibile agli sviluppatori delle rispettive piattaforme.

1.3 Applicazioni Location-Aware

Le applicazioni definite *Location-Aware* sono tutte quelle che basano parte delle loro funzionalità sui servizi di localizzazione per identificare la posizione dell'utente. L'utilizzo di queste applicazioni è ormai esteso nella maggior parte dei settori poiché offre il vantaggio di fornire informazioni e contenuti all'utente basandosi sulla sua posizione e sulle aree geografiche che si trovano nei suoi paraggi, potendogli ad esempio fornire servizi più semplici e veloci per la ricerca di un luogo o di una persona.

[9] Magic Leap One, `https://www.magicleap.com/magic-leap-one`.
[10] ARKit, `https://developer.apple.com/arkit/`.
[11] ARCore, `https://developers.google.com/ar/`.

I dettagli sulla posizione geografica di un device possiamo ottenerli utilizzando tecnologie differenti, dalle infrastrutture delle reti cellulare agli access-point wireless fino ai dati satellitari. Grazie al GPS integrato in tutti gli attuali smartphone, determinare le coordinate relative alla posizione di un utente è relativamente semplice.

Queste tecnologie per l'identificazione della posizione, soprattutto nell'ambito mobile, sono in continuo sviluppo, in quanto la precisione è ancora approssimativa con un errore che varia tra i 5 ed i 20 metri. Per esempio se ci troviamo all'interno di un edificio oppure nelle vie di un centro urbano il segnale GPS sarà debole, mentre se ci troviamo all'aperto in una zona con pochi edifici o pochi alberi il segnale ricevuto sarà migliore [48].

Per sviluppare questo tipo di applicazioni è necessario sfruttare delle APIs che forniscano informazioni geografiche, ad esempio possiamo utilizzare i *Google Play Services*[12] per Android oppure il *MapKit*[13] per iOS.

1.4 POI: Point Of Interest

Possiamo definire un *Punto di Interesse*[14] come un punto specifico nel mondo che qualcuno può trovare utile o interessante. *W3C* fornisce una descrizione più specifica, definendolo come un insieme di termini geografici accoppiati ed interconnessi che comprende la definizione di 3 principali componenti [49]:

- *Locations*: inteso come un costrutto geografico, un punto fisico fissato sulla superficie della Terra descritto da latitudine, longitudine ed altitudine;

- *POIs*: inteso come un costrutto umano che descrive cosa è possibile trovare in una specifica *Location*; di solito un *POI* è composto da vari attributi come ad esempio un nome, una *Location* ed una categoria

[12]Google API, `https://developers.google.com/android/guides/overview`.
[13]Apple MapKit, `https://developer.apple.com/documentation/mapkit`.
[14]POI, `https://en.wikipedia.org/wiki/Point_of_interest`.

e non è detto che sia un punto fisso, ma si può muovere tra diverse *Locations*;

- *Places*: inteso anch'esso come un costrutto umano ma con un livello di granularità più grossolano, utilizzato di solito per definire continenti, stati, città oppure piccoli distretti; un *Place* può contenere uno o più *POIs*, ad esempio nel caso di una città, oppure può combaciarvi, ad esempio il Parco Sempione di Milano può essere considerato sia un *Place* che un *POI*.

1.5 Stato dell'Arte e Lavori Correlati

Negli ultimi anni tutte le compagnie leader mondiali nel settore, come Microsoft, Google, Apple e Facebook, hanno iniziato ad aumentare notevolmente la quantità di capitale investito nella ricerca sulla Realtà Aumentata, accelerandone così lo sviluppo, la crescita e la diffusione. In particolare sono molte le librerie ed i tools per lo sviluppo di applicazioni in AR che ci offrono, ad esempio, funzionalità accurate di motion tracking con 6DOF (*Six Degree Of Freedom*), di renderizzazione contenuti 3D e di riconoscimento di immagini bidimensionali o di oggetti tridimensionali.

Tutte queste funzionalità possono essere combinate con le *applicazioni Location-Aware*, fornendo così contenuti in AR in base alla posizione dell'utente. Possiamo quindi suddividere le applicazioni in AR in due principali categorie:

1. *Marker-based*, che si occupano principalmente di image recognition per identificare alcuni *marker* specifici come codici QR, immagini o particolari pattern;

2. *Markerless*, anche dette *location-based*, che non hanno bisogno di riconoscere nessun *marker* ma offrono esperienze sulla base dei dati ricevuti dalla fotocamera, dal GPS e dai sensori dello smartphone.

Le applicazioni in AR *location-based* hanno tratto un enorme vantaggio dalla tecnologia *SLAM* (Simultaneous Localization And Mapping), che fornisce la possibilità di comprendere le caratteristiche dell'ambiente in cui ci troviamo mappandolo in modo più accurato al fine di posizionare correttamente oggetti virtuali in un contesto reale, e dall'evoluzione e diffusione degli smartphone e dei sensori al loro interno.

Grazie a tutto ciò, la Realtà Aumentata è ormai ampiamente utilizzata in molti settori, come ad esempio nel settore medicinale [50], in quello manifatturiero ed ingegneristico [51], nell'educativo [52] e nel geologico [53], continuando a diffondersi velocemente in molti altri [54]. Al giorno d'oggi l'AR sta iniziando a far parte delle nostre vite e delle nostre routine grazie alle applicazioni mobile ed al suo utilizzo nei Social Network e nelle campagne pubblicitarie (Figura 1.16): per esempio su Instagram possiamo applicare alle nostre *Storie* dei filtri che sfruttano la Realtà Aumentata oppure su Snapchat possiamo creare dei video in cui contenuti virtuali vengono sovrapposti alle immagini che catturiamo con la nostra fotocamera, mentre Nike ha sfruttato l'AR per la vendita di alcune sneakers in produzione limitata [55].

Per sopperire alla necessità ed alla maggiore richiesta di poter sviluppare applicazioni in AR *location-based* sono nati diversi progetti nel corso degli ultimi anni. Uno tra questi è il progetto *AREA* (Augmented Reality Engine Application) [59], attualmente aggiornato alla seconda versione *AREAv2* [60]: un kernel che permette di sviluppare queste applicazioni individuando *POIs* predefiniti che vengono successivamente forniti all'utente in modo interattivo, ad esempio facendo in modo che sia possibile ottenere informazioni toccandoli.

Il primo Marzo 2018, dopo tre anni di ricerche e sviluppi, Google annuncia la chiusura del *Project Tango* a favore della nuova piattaforma *ARCore*[15] per lo sviluppo di applicazioni in AR per Android, in modo tale da poter tenere il passo con l'uscita di *ARKit*[16] da parte di Apple.

[15] ARCore, `https://developers.google.com/ar/`.
[16] ARKit, `https://developer.apple.com/arkit/`.

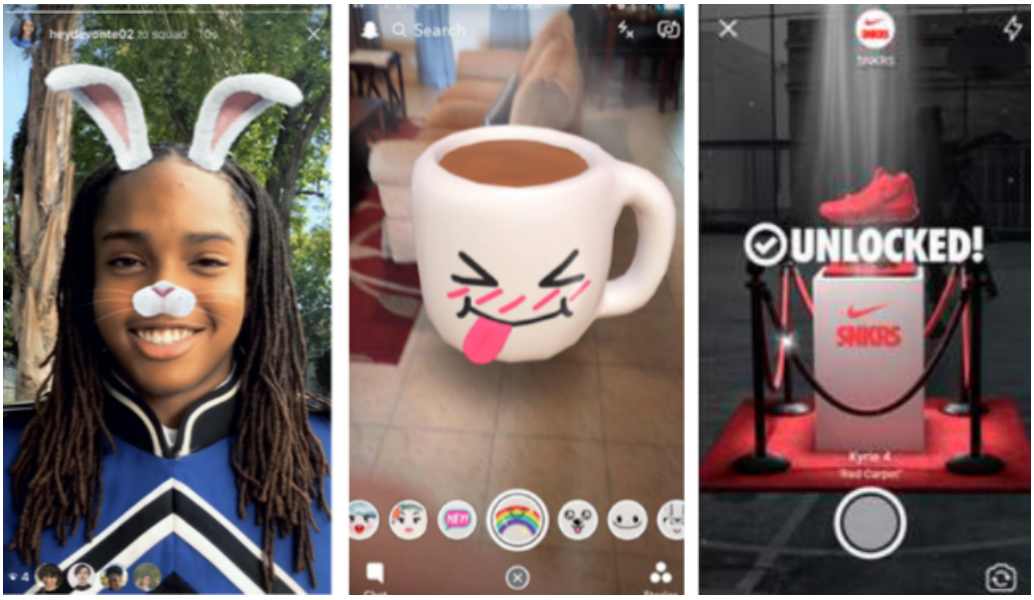

Figura 1.16: Realtà Aumentata applicata nei Social Network e nel marketing. Partendo da sinistra: Instagram [56], Snapchat [57] e Nike [58].

Attualmente esistono moltissimi tools e piattaforme per lo sviluppo di applicazioni, mobile e non, che sfruttino la Realtà Aumentata, ognuno dei quali fornisce diverse funzionalità e ci permette di realizzare varie tipologie di prodotti. Nell'ultimo anno Facebook ha rilasciato *AR Studio*[17], mentre Snap Inc. ha lanciato sul mercato *Lens Studio*[18], entrambi strumenti di sviluppo per utenti che cercano di creare esperienze in AR per la distribuzione sulla relativa piattaforma. *Magic Leap* rilascia ufficialmente il *Magic Leap One*[19] e tutto il necessario per potervi creare esperienze in AR, per non parlare di altre piattaforme diffuse sul mercato come *Amazon Sumerian*[20], *Vuforia*[21], *Mapbox*[22] e *Wikitude*[23].

[17]Spark AR Studio, `https://www.sparkar.com/ar-studio/`.
[18]Lens Studio, `https://lensstudio.snapchat.com/`.
[19]Magic Leap One, `https://www.magicleap.com/magic-leap-one`.
[20]Amazon Sumerian, `https://aws.amazon.com/it/sumerian/`.
[21]Vuforia, `https://www.vuforia.com/`.
[22]Mapbox, `https://www.mapbox.com/augmented-reality/`.
[23]Wikitude, `https://www.wikitude.com/geo-augmented-reality/`.

Durante l'*AWE18* (Augmented World Expo 2018) sono state moltissime le società e le startup che hanno annunciato la creazione di tools per lo sviluppo di esperienze in AR, come ad esempio *Qualcomm* [61], *ScopeAR* [62] oppure la Niantic, che ha dichiarato l'imminente rilascio della *Niantic Real World Platform* [63], la piattaforma utilizzata per sviluppare *Pokémon Go*.

Tra le piattaforme citate in precedenza, ce ne sono alcune che ci permettono di creare esperienze in Realtà Aumentata legate a punti di interesse. La migliore fra tutte, attuale leader di mercato nel settore, è sicuramente *Wikitude*[24], piattaforma che sfrutta tecnologie proprietarie per fornire al pubblico un SDK cross-platform con cui poter lavorare su dati georeferenziati, rendendo molto più semplice sia il processo di reperire *POIs* sia quello di posizionarci dei contenuti in Realtà Aumentata.

Un'altra piattaforma è *Mapbox*[25], con cui è possibile realizzare applicazioni tramite Unity sfruttando i *POIs* presenti nel relativo SDK *Map*, potendo così cercare, aggiungere e modificare contenuti relativi ad un *POI* sulla mappa direttamente dall'editor di Unity.

Anche *BlippAR* fornisce un SDK che sfrutta una tecnologia proprietaria per posizionare contenuti in AR *location-based* chiamata *Urban Visual Positioning System*, offrendo così la possibilità agli sviluppatori di realizzare questo tipo di esperienze nelle loro applicazioni [64].

Tra i frameworks e le piattaforme che permettono lo sviluppo di contenuti in AR, ce ne sono alcune che pur essendo diffuse non offrono la possibilità di posizionare contenuti in particolari *POIs* nativamente. Ad esempio *EasyAR* e *Vuforia* sono piattaforme diffuse nello sviluppo di applicazioni AR, ma nell'acquisto del relativo SDK non sono inclusi dati georeferenziati o mappe. Diventa necessario quindi appoggiarsi a servizi esterni, come ad esempio le APIs di Google citate in precedenza, per poter ottenere dati relativi alla posizione dell'utente. Seguendo questa logica sono stati sviluppati vari lavori,

[24]Wikitude, `http://www.wikitude.com/external/doc/documentation/`.

[25]Mapbox, `https://www.mapbox.com/unity-sdk/maps/examples/poi-placement/`.

anche da amatori, per poter unire queste due funzionalità:

- *AR-POI*[26], una libreria solo per sistemi iOS che sfrutta *ARKit* e le APIs di Google per cercare posti attorno all'utente e mostrarglieli utilizzando la Realtà Aumentata;
- *ARCore-Location*[27], una libreria solo per sistemi Android che cerca di integrare *ARCore* e le APIs di Google;
- *ARKit-CoreLocation*[28], altra libreria per soli sistemi iOS che combina *ARKit* ad una seconda libreria chiamata *CoreLocation* per poter posizionare contenuti in AR utilizzando coordinate reali;
- *ARAround*[29], libreria per Android per visualizzare *POIs* attorno all'utente in Realtà Aumentata;
- *Android Augmented Reality GPS Points*[30], una libreria Java che sfrutta l'AR per visualizzare il nome, l'altitudine e la distanza di un *POI* dall'utente.

La maggior parte di queste librerie sviluppate sono molto limitate poiché offrono solo la possibilità di visualizzare alcune delle informazioni relative a punti di interesse in modo statico. La diffusione di questi progetti ci fa però capire quanto il mondo della Realtà Aumentata sia ormai alla portata di tutti, continuando ad evolversi con una velocità impressionante. Chiunque, dalla grande compagnia leader del settore al piccolo sviluppatore nella comodità di casa sua, può sviluppare un'applicazione che combini l'AR alle potenzialità degli attuali smartphone grazie alle innumerevoli piattaforme lanciate sul mercato.

[26] AR-POI, `https://github.com/inorganik/AR-POI`.
[27] ARCore-Location, `https://www.appoly.co.uk/arcore-location/`.
[28] ARKit-CoreLocation, `https://github.com/ProjectDent/ARKit-CoreLocation`.
[29] ARAround, `https://github.com/mchochlov/ARAround`.
[30] Android Augmented Reality GPS Points, `https://github.com/AlexandreLouisnard/android-augmented-reality-gps-points`.

Capitolo 2

Analisi del problema

2.1 Problema di partenza

Le ricerche e gli studi effettuati in questa tesi nascono in sinergia con l'azienda *Comuni-Chiamo S.r.l.*[1] di Borgo Panigale (BO) per poter migliorare l'esperienza degli utenti nell'utilizzo della propria applicazione.

Comuni-Chiamo si occupa della realizzazione di soluzioni in cloud per gestire al meglio la città ed innovare il Comune: dall'App per informare i cittadini al software per gestire le segnalazioni. In particolare, da un lato i cittadini possono sfruttare la comoda app per segnalare malfunzionamenti o danni all'interno del proprio Comune di appartenenza, dall'altro lato il Comune può gestire tutte le segnalazioni ricevute fornendo la possibilità agli operatori comunali di raggiungere i luoghi segnalati. É proprio quest'ultima parte dell'applicativo quella su cui si voleva migliorare l'esperienza dell'utente, in particolare si voleva facilitare e velocizzare la ricerca del danno segnalato da parte dell'operatore comunale: può capitare che la segnalazione sia arrivata con una posizione GPS imprecisa, che non ci sia una fotografia allegata del danno oppure che la descrizione non sia chiara ed esaustiva, tutte motivazioni che rendono più difficile il lavoro dell'operatore comunale nel trovare, controllare e riparare il danno.

[1] Comuni-Chiamo S.r.l., `https://comuni-chiamo.com/`.

2.2 Astrazione del problema

Accertato il fatto che questo problema esisteva, in almeno una azienda, nella fase di analisi iniziale abbiamo cercato di astrarre il problema il più possibile, in modo tale da poter trovare una o più soluzioni applicabili non solo nel contesto specifico di questa azienda. Astraendo questo problema sono stati individuati due fattori chiave su cui poter lavorare: i *POIs* ed il *Tempo*.

Il *POI* (Point Of Interest) può essere *statico*, nel caso in cui abbia una *Location* fissa, o *dinamico*, nel caso in cui invece si muova tra diverse *Location*, così come anche il *Tempo* può essere *statico* o *dinamico*. Questi due fattori si influenzano a vicenda formando 4 diversi scenari possibili su cui poter lavorare che vengono descritti in Tabella 2.1.

		Tempo	
		Statico	**Dinamico**
POI	**Statico**	La *Location* del *POI* è una sola, è fissa geograficamente ed è persistente nel tempo.	La *Location* del *POI* è una sola ed è fissa geograficamente, ma non è persistente nel tempo.
	Dinamico	Il *POI* ha più di una *Location* e si muove tra esse, ma è persistente nel tempo.	Il *POI* ha più di una *Location* e si muove tra esse e non è persistente nel tempo.

Tabella 2.1: Descrizione dei 4 possibili scenari su cui poter lavorare.

Per dare una maggiore spiegazione, quando parliamo di *POI Statico con Tempo Statico* ci riferiamo ad un luogo indicato da una particolare coordinata GPS che non cambia e che non sparirà con il passare del tempo, ad esempio un monumento oppure una piazza, mentre quando parliamo di *POI Dinamico con Tempo Dinamico* intendiamo un punto di interesse la cui coordinata GPS cambia e che verrà cancellato con il passare del tempo, ad esempio il *POI*

potrebbe essere un indicatore da seguire per raggiungere un punto di incontro ed una volta che viene raggiunto il *POI* può essere cancellato. In Tabella 2.2 possiamo vedere alcuni esempi pratici di questi scenari.

		Tempo	
		Statico	**Dinamico**
POI	**Statico**	Visita dei monumenti di una città. Visualizzazione di aree di sgambatura cani. Ricerca di hotspot Wi-Fi gratuiti. Ricerca di aree verdi e parchi.	Navigazione basata su punti di riferimento. Ritrovare l'auto parcheggiata. Realizzare una caccia al tesoro. Soccorrere persone in pericolo.
	Dinamico	Ritrovare una persona in mezzo alla folla. Condivisione della posizione live.	Visita guidata di una città seguendo un *POI*. Raggiungere un luogo seguendo un *POI*.

Tabella 2.2: Possibili scenari applicativi dei 4 casi analizzati.

L'astrazione del problema iniziale ci ha quindi portato ad ottenerne uno più ampio su cui poter fare maggiori analisi, ovvero il problema legato a come poter migliorare le esperienze degli utenti che effettuano operazioni legate a punti di interesse.

2.3 Analisi di possibili soluzioni

Una volta astratto, prima di passare all'analisi di una o più soluzioni, è stato necessario limitare il problema ad un contesto più specifico: l'ambito

delle applicazioni mobile, scelta fondamentale che ci ha permesso di focalizzare le ricerche e le analisi su un minor numero di problematiche. A tal proposito, l'analisi di possibili soluzioni si è concentrata sulla ricerca di quali sono le tecnologie in ambito mobile esistenti e quali di queste sono applicabili per questo particolare problema preso in esame. Tra le tecnologie esistenti vi è la Realtà Aumentata che, come introdotto nel capitolo precedente, sta prendendo sempre più piede nella vita quotidiana degli utenti migliorando alcune delle loro routine.

Innanzitutto sono state analizzate alcune piattaforme o applicazioni sul mercato che si scontrano con questa tipologia di problema ma non sfruttano la Realtà Aumentata:

- **Comuni-Chiamo**[2]: come anticipato precedentemente, l'operatore comunale sfrutta l'app per raggiungere i luoghi in cui sono stati segnalati dei danni; le segnalazioni rappresentano i *POIs Statici* mentre in questo caso abbiamo *Tempo Dinamico*, poiché dopo aver riparato il danno segnalato il relativo *POI* non è più necessario;

- **Life360**[3]: permette di rimanere in contatto con le persone più care aggiungendole ai propri contatti all'interno dell'app; una volta aggiunte, è possibile condividere la propria posizione GPS con loro e visualizzarne la posizione in tempo reale; i *POIs* sono rappresentati dalla posizione GPS dei contatti aggiunti e sono *Dinamici* in quanto la condivisione è in tempo reale, per cui il contatto potrebbe essere in movimento, mentre il *Tempo* è *Statico* perché, salvo imprevisti, la posizione GPS di un contatto è permanente nel tempo;

- **i-React**[4]: è la prima applicazione a livello europeo che si occupa della gestione dei dati relativi ad emergenze o disastri ricevuti da risorse multiple; in questo caso saranno proprio i dati ricevuti a contenere la

[2]Comuni-Chiamo, `https://comuni-chiamo.com/products`.
[3]Life360, `https://www.life360.com/`.
[4]i-React, `http://www.i-react.eu/`.

posizione GPS di un'emergenza e siamo nel caso di *POI Statico* con *Tempo Dinamico*;

- **PulsePoint**[5]: applicazione utilizzata principalmente in America che permette ai cittadini di ricevere una notifica quando qualcuno nei paraggi invia una richiesta di soccorso per un arresto cardiaco, in modo tale che il primo soccorso sia tempestivo; anche in questo scenario ci troviamo nel caso di *POI Statico*, ovvero il paziente da soccorrere, con *Tempo Dinamico*.

La Realtà Aumentata potrebbe essere integrata in tutte queste piattaforme come possibile miglioramento della *User Experience* grazie alla possibilità di annotare il mondo che ci circonda. Non solo, utilizzare la Realtà Aumentata porta vantaggi anche per quanto riguarda la sicurezza degli utenti, i quali, ad esempio, invece di guardare una mappa sul proprio smartphone per raggiungere un *POI* potrebbero inquadrare il mondo che li circonda e raggiungere la propria destinazione attraverso le indicazioni visualizzate direttamente sulle immagini in tempo reale prese dalla fotocamera, in modo tale da avere una maggiore coscienza e percezione di ciò che li circonda in quel momento.

Di app che sfruttano la Realtà Aumentata in scenari molto simili ne esistono ancora poche, per esempio *Find Your Car with AR*[6] ti permette di ritrovare dove hai parcheggiato l'automobile, mentre *ARoundU*[7] ti mostra i monumenti o le attrazioni della città in cui ti trovi, ma ci bastano questi pochi esempi per capire come la Realtà Aumentata può essere impiegata per fornire all'utente un'esperienza migliore con cui poter svolgere le sue mansioni.

Per tutti questi motivi e per i potenziali vantaggi che ne derivano dal suo utilizzo, tra le possibili soluzioni da implementare per offrire esperienze migliori agli utenti si è scelto di mettere l'AR al centro della soluzione.

[5]PulsePoint, `https://www.pulsepoint.com/`.

[6]Find Your Car with AR, `https://itunes.apple.com/us/app/find-your-car-with-ar/id370836023?mt=8`.

[7]ARoundU, `https://itunes.apple.com/us/app/around-u/id1317512654?mt=8`.

2.4 Tecnologie esistenti analizzate

Come anticipato nel capitolo precedente, le tecnologie esistenti per lo sviluppo di app che sfruttino la Realtà Aumentata sono tante ed in continuo aumento. Tra queste tecnologie ne sono state analizzate alcune nel dettaglio, in particolare quelle che offrivano tutte le funzionalità necessarie per la creazione di contenuti in AR legati a punti di interesse. Tra queste tecnologie analizzate troviamo **Wikitude**, **Mapbox**, **ARCore**, **ARKit** e **Kudan AR**.

2.4.1 Wikitude

Attuale leader di mercato nel settore dello sviluppo di applicazioni in Realtà Aumentata, fornisce supporto sia per Android che per iOS ed implementa funzionalità di image recognition, image tracking, riconoscimento di oggetti 3D ed offre la possibilità di sfruttare dati georeferenziati. Uno dei principali punti di forza di questo SDK è la tecnologia *SLAM* proprietaria sfruttata per il tracking degli oggetti, la quale permette un posizionamento accurato nello spazio circostante.

Oltre a tutte queste funzionalità, durante quest'anno *Wikitude* ne ha introdotta un'altra chiamata *SMART*: Seamless AR Tracking. *SMART* unisce in un solo SDK tutto il necessario per poter sviluppare applicazioni mobile sfruttando al 100% le potenzialità del device grazie alla fusione di *ARCore*, *ARKit* e dell'engine *SLAM* proprietario, fornendo quindi la possibilità di non dover più scrivere codice nativo specifico per ogni piattaforma per cui si vuole sviluppare.

Tutte queste funzionalità hanno però un costo elevatissimo, che è il principale svantaggio di questo SDK. Come è possibile vedere in Figura 2.1 esistono diverse tipologie di licenza in base alla quantità di funzionalità che si vogliono utilizzare: da un minimo di 499€ ad un massimo di 2490€ una tantum.

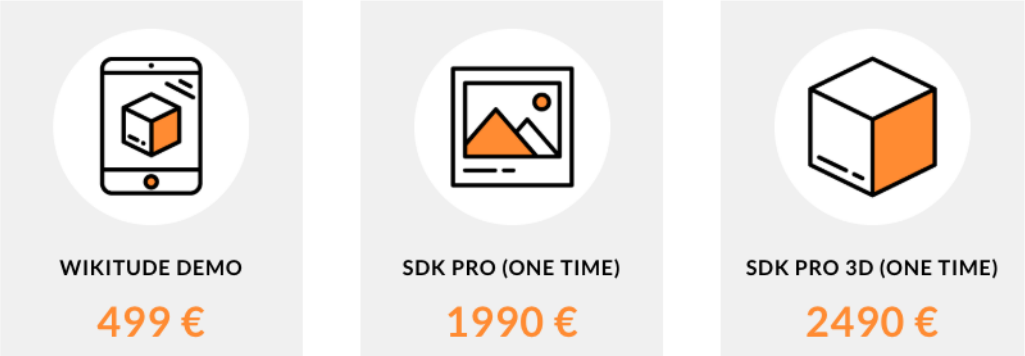

Figura 2.1: Prezzo delle licenze di utilizzo di *Wikitude SDK* [65].

2.4.2 Mapbox

Mapbox è una piattaforma per lo sviluppo di applicazioni web e mobile legate a dati geolocalizzati. Tra le funzionalità che offre troviamo l'editor per creare o modificare delle mappe e degli ambienti 3D, una grande quantità di dati geolocalizzati, dati in tempo reale sul traffico e la navigazione tra due coordinate. Tutte queste funzionalità sono integrate allo sviluppo di applicazioni AR tramite due SDK:

- *Apple SceneKit SDK*, per lo sviluppo di applicazioni su iOS;
- *React Native AR SDK*, per lo sviluppo di applicazioni cross-platform.

Grazie a questi due SDK è possibile realizzare app in AR sia per iOS che per Android utilizzando le funzionalità native fornite dai vari sistemi.

Come nel caso di *Wikitude*, anche *Mapbox* non è gratuito ma offre un piano chiamato *Pay-as-you-go*, in cui lo sviluppatore paga in base al numero di utilizzi di *Mapbox* che vengono fatti dagli utenti delle sue applicazioni, oppure un piano *Commercial* che prevede un pagamento di 499€ al mese per avere l'accesso a tutte le funzionalità offerte dalla piattaforma.

2.4.3 Kudan AR

Tra le varie funzionalità offerte da questa piattaforma per lo sviluppo di applicazioni in Realtà Aumentata, possiamo identificare la tecnologia pro-

prietaria *SLAM* come il principale punto di forza. Grazie a questa tecnologia, *Kudan AR* offre un approccio al tracking ed al mapping degli oggetti e della scena dalle elevate performance, potendo posizionare gli oggetti in modo molto preciso ed accurato. Grazie a questo vantaggio, questa piattaforma si presta perfettamente allo sviluppo di applicazioni in AR *location-based*.

La piattaforma offre un SDK che permette lo sviluppo di applicazioni cross-platform, supportando un gran numero di dispositivi mobile, ed offre tre tipi di licenze:

- *AR Indie*: gratuita, ma solo per utilizzo non commerciale;
- *AR Business*: ha un costo di 1000€ l'anno ed offre tutte le funzionalità a patto che i ricavi ottenuti dalle applicazioni sviluppate con questo SDK non superino una certa soglia; questo tipo di licenza è adatto per sviluppo di applicazioni ad uso istituzionale oppure no-profit;
- *AR Enterprise*: offre tutte le funzionalità senza nessun limite ad un prezzo che va concordato in base alle proprie necessità.

2.4.4 ARCore

Questo framework per lo sviluppo di applicazioni in AR proprietario di Google nasce come naturale evoluzione del *Project Tango*, con la differenza che permette di sviluppare applicazione in Realtà Aumentata senza bisogno di nessun hardware esterno al dispositivo.

Tra le funzionalità offerte da questo SDK ci sono il motion tracking, il riconoscimento dell'ambiente, il riconoscimento di superfici sia orizzontali che verticali e la capacità di stimare la quantità di luce nell'ambiente circostante. *ARCore* inoltre si presta bene allo sviluppo di applicazioni in AR *location-based* grazie alla capacità di posizionare correttamente ed in modo accurato contenuti virtuali nel mondo circostante. *ARCore* offre anche la possibilità di utilizzare *Sceneform*, una libreria di alto livello che permette di realizzare scene in Realtà Aumentata senza dover utilizzare OpenGL direttamente,

rendendo così lo sviluppo di questo tipo di applicazioni ancora più semplice ed accessibile agli sviluppatori.

A differenza dei precedenti SDK, *ARCore* offre meno funzionalità ma è nativamente integrato nei dispositivi Android ed è completamente gratuito, sia per utilizzi non commerciali che per utilizzi commerciali. Il principale svantaggio di questa piattaforma è il limitato numero di dispositivi supportati, numero che però è in continuo aumento e che molto probabilmente non rappresenterà più un problema in quanto, da qui ad un anno, la maggior parte dei device saranno supportati.

2.4.5 ARKit

ARKit è l'SDK ufficiale di Apple per lo sviluppo di applicazioni mobile in AR, introdotto in iOS 11. Questo SDK offre tutte le funzionalità necessarie per lo sviluppo di applicazioni in AR *location-based*: dalla capacità di comprendere la scena e di stimare la quantità di luce nell'ambiente all'utilizzo del *Visual Inertial Odometry* (VIO), ovvero la tecnologia realizzata per mappare accuratamente l'ambiente circostante ottimizzando il tracking degli oggetti, il tutto unito a smartphone molto potenti dotati di una vasta gamma di sensori di ultima generazione. La piattaforma supporta il riconoscimento di piani sia orizzontali che verticali, offrendo così la possibilità di posizionare contenuti virtuali in modo molto accurato su entrambe le tipologie di superfici.

Proprio come *ARCore*, questo SDK è nativamente integrato nei dispositivi Apple ed è completamente gratuito.

2.4.6 Tabella riassuntiva

Tutti questi SDK analizzati finora offrono la possibilità e le funzionalità per realizzare un'applicazione in AR *location-based* di ottima qualità. Possiamo quindi riassumere tutte le loro principali caratteristiche in Tabella 2.3.

	Wikitude	**ARCore/ ARKit**	**Mapbox**	**Kudan AR**
Piattaforme	Cross-platform	Android/iOS	Cross-platform	Cross-platform
Pro	*SLAM* proprietario, image recognition e dati georeferenziati.	Gratuiti, nativi ed in continuo aggiornamento.	Editor per le mappe, dati georeferenziati e navigazione.	*SLAM* proprietario, elevate performance.
Contro	Prezzo elevato.	Non permettono sviluppo cross-platform, sono pochi i device supportati.	Prezzo elevato, non dispone di *SLAM* proprietario.	Prezzo elevato, non offre dati georeferenziati.

Tabella 2.3: Caratteristiche delle principali tecnologie analizzate.

A seguito delle analisi effettuate su queste tecnologie esistenti sul mercato si è deciso di utilizzare *ARCore* per lo sviluppo della soluzione proposta. Questa scelta è stata dettata dal fatto che gli SDK offerti da *Wikitude*, *Mapbox* e *Kudan* hanno un costo di utilizzo troppo elevato per gli scopi di questa tesi, mentre il non utilizzo di *ARKit* a favore di *ARCore* è dovuto puramente ad una scelta personale, in quanto entrambi offrono le stesse funzionalità sulle relative piattaforme.

2.5 ARCore e Sceneform

Fondamentalmente *ARCore* si basa sul tracking del device, ovvero sulla posizione dello smartphone e sui movimenti che vengono fatti, per la registrazione di tutte le informazioni sul mondo reale che verranno poi utilizzate per la costruzione del mondo in Realtà Aumentata. Tutte le funzionalità offerte da questo SDK possono essere combinate con *Sceneform*, una serie di APIs per il rendering di scene e contenuti 3D in AR in modo semplice e veloce, senza dover apprendere OpenGL.

Di seguito illustreremo i concetti e le funzionalità chiave su cui si basano *ARCore* e *Sceneform* per poter sviluppare applicazioni in Realtà Aumentata.

2.5.1 Motion Tracking

Una delle funzionalità principali fornite da *ARCore* per lo sviluppo di applicazioni in AR è proprio il *motion tracking* del device, il quale viene fatto sfruttando la fotocamera dello smartphone per identificare e tracciare nel tempo dei **feature points**, ovvero i punti che hanno caratteristiche interessanti e rilevanti al fine di collezionare il maggior numero di informazioni sul mondo reale. La tecnologia generale che permette di realizzare il tracking è il *Simultaneous Localization and Mapping* (SLAM), processo descritto anche precedentemente che colleziona dati da fotocamera, accelerometro, giroscopio ed altri sensori al fine di comprendere la composizione del mondo reale che si sta inquadrando. In particolare, il processo utilizzato da *ARCore* è chiamato **Concurrent Odometry and Mapping** (COM)[8] che, essenzialmente, non fa altro che capire dov'è posizionato il device rispetto al mondo reale che lo circonda catturando le distinte caratteristiche dell'ambiente, come ad esempio lo spigolo di una sedia, le fughe del pavimento, le venature del legno, un portapenne o qualunque altro elemento che potenzialmente crea una sorta di contrasto con l'ambiente circostante.

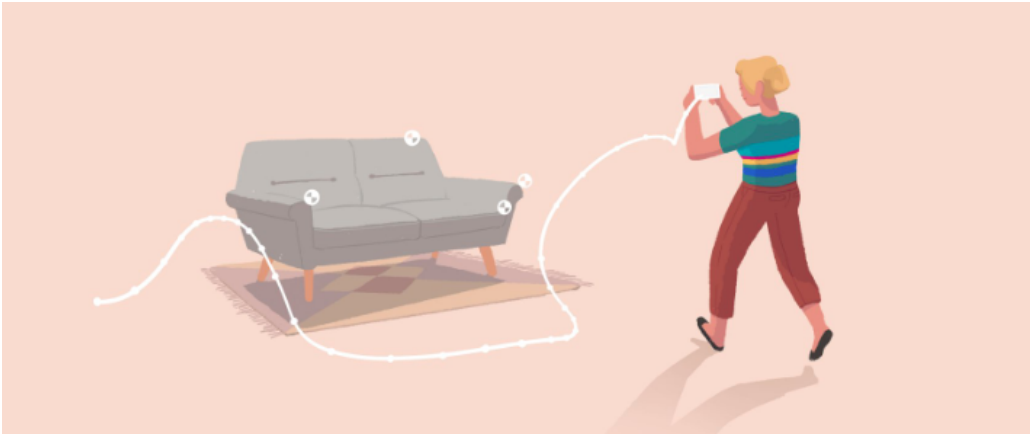

Figura 2.2: Riconoscimento di *feature points* tramite *motion tracking* [66].

[8]ARCore: Concurrent Odometry and Mapping, `https://patents.google.com/patent/US20170336511A1/en`.

L'utilizzo del *COM* per la creazione di una conoscenza del mondo reale è possibile grazie all'hardware che risiede negli smartphone attuali, in particolare:

- la *fotocamera*, che fornisce un flusso di informazioni in tempo reale sull'ambiente in cui dover creare contenuti virtuali, informazioni su cui è possibile applicare tecniche di machine learning, image processing e computer vision al fine di produrre immagini di alta qualità e mappe spaziali per la creazione di applicazioni in AR;
- l'*accelerometro*, sensore che misura l'accelerazione subita dal device, in modo tale da ottenere tutte le informazioni sugli spostamenti dell'utente nello spazio;
- il *giroscopio*, sensore che misura l'orientazione e la velocità angolare per capire in che modo viene ruotato il device dall'utente.

Unendo tutte le informazioni ricevute dalla fotocamera e da questi sensori, *ARCore* è in grado di determinare la cosiddetta **Pose**[9] del device, ovvero la sua posizione e il suo orientamento relativi al mondo reale nel tempo.

2.5.2 Riconoscimento dell'ambiente

Il riconoscimento di *feature points* non viene utilizzato solamente per determinare la *Pose* del device, ma anche per riconoscere in modo accurato gli oggetti e le superfici che costituiscono l'ambiente che lo circonda. In particolare, per il riconoscimento di superfici *ARCore* cerca dei gruppi di *feature points* che sembrano risiedere su una stessa superficie orizzontale o verticale, creando così dei piani virtuali su cui poter posizionare contenuti digitali in modo accurato e preciso.

ARCore è anche in grado di determinare i contorni di ogni superficie che viene riconosciuta, in modo tale che sia possibile specificare alcune azioni

[9] ARCore: Pose, `https://developers.google.com/ar/reference/java/com/google/ar/core/Pose`.

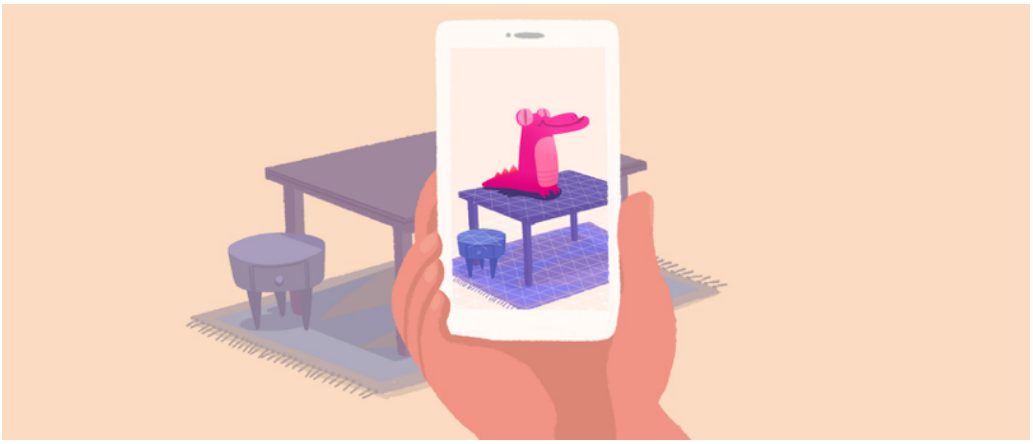

Figura 2.3: Posizionamento di un contenuto virtuale su una superficie riconosciuta nell'ambiente inquadrato dalla fotocamera [66].

legate ai contenuti virtuali posizionati, ad esempio è possibile specificare che un particolare oggetto debba risiedere sempre sul piano orizzontale e mai su un piano verticale adiacente.

Come la maggior parte dei framework per l'AR, anche per *ARCore* diventa problematico riconoscere superfici ed oggetti senza un adeguato livello di contrasto, per esempio sarà molto difficile riconoscere un muro bianco oppure un pavimento completamente scuro.

2.5.3 Stima della luce

Per incrementare il livello di realismo della scena creata in Realtà Aumentata, *ARCore* offre persino la funzionalità di **light estimation**[10], ovvero è in grado di capire l'intensità e la quantità di luce nell'ambiente circostante semplicemente analizzando le immagini ricevute dalla fotocamera del device. Basandosi su queste informazioni è in grado di applicare le giuste correzioni al colore ed alla luminosità della scena creata, mostrando i contenuti virtuali con la stessa quantità di luce degli oggetti reali che circondano tale contenuto.

[10] ARCore: Light Estimation, `https://developers.google.com/ar/reference/java/com/google/ar/core/LightEstimate`.

Al fine di fornire una migliore esperienza all'utente, dato che la *Pose* del device può cambiare con il passare del tempo, *ARCore* si assicura che tutti i contenuti virtuali siano renderizzati in modo corretto tenendo in considerazione anche lo spostamento del punto da cui viene inquadrata la scena.

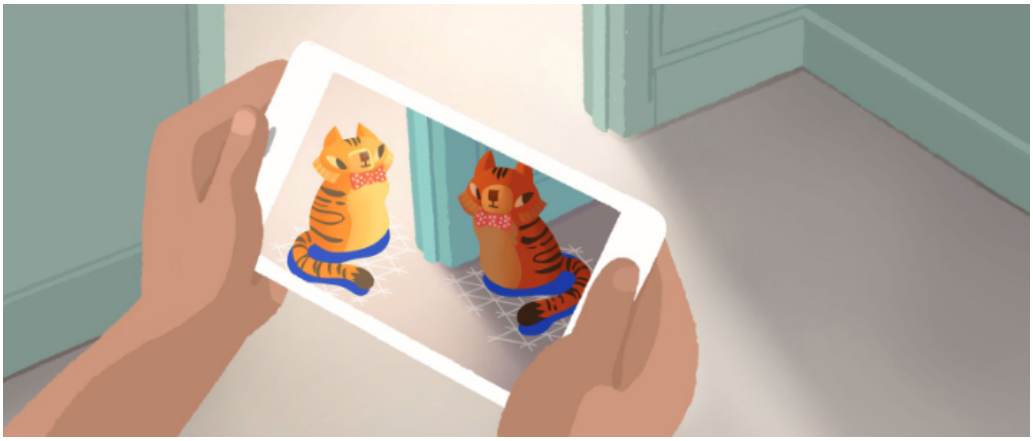

Figura 2.4: Colori adeguati alla quantità di luce dell'ambiente [66].

2.5.4 Interazione con l'utente

Uno dei tanti fattori importanti quando si sviluppano applicazioni mobile in AR è sicuramente la gestione dell'interazione con l'utente. A tal scopo, *ARCore* utilizza il processo chiamato **hit-testing**[11] per determinare se un'azione dell'utente sullo schermo del device corrisponde ad un'azione nella scena. Tramite l'*hit-testing* viene proiettato un raggio dalla fotocamera del device nella scena, la cui successiva elaborazione permetterà di ottenere le informazioni sulle superfici e sui *feature points* intersecati.

Grazie a questa funzionalità possiamo, ad esempio, ottenere la coordinata (x, y) nella scena che corrisponde ad un tap dell'utente sullo schermo del device e capire se quell'interazione ha azionato una particolare dinamica

[11] ARCore: Hit-testing, `https://developers.google.com/ar/reference/java/com/google/ar/core/HitResult`.

legata ad un contenuto virtuale presente sulla scena, offrendo all'utente la possibilità di selezionare o interagire con tutti i contenuti virtuali.

2.5.5 Contenuti virtuali: Anchor e Trackable

Come detto in precedenza, *ARCore* è in grado di capire la *Pose* del device all'interno del mondo reale. Questa *Pose* può cambiare nel tempo, cambiamento dovuto sia ai movimenti dell'utente che alla capacità di *ARCore* di migliorare la propria conoscenza sull'ambiente circostante. Proprio a causa di questi continui cambiamenti ed aggiornamenti della *Pose*, quando vogliamo posizionare dei contenuti virtuali all'interno della nostra scena e vogliamo far sì che rimangano in una particolare posizione, abbiamo bisogno di definire un'*ancora*, definita appunto da *ARCore* con il termine di **Anchor**[12].

Le *Anchor* descrivono una posizione ed una orientazione fissi nel mondo reale, in modo tale che sia possibile creare dei contenuti che rimangano nello stesso punto anche mentre l'utente si muove attorno ad essi per esplorare la scena. Se vogliamo ad esempio creare un contenuto virtuale quando l'utente effettua un tap sullo schermo, possiamo utilizzare la coordinata (x, y) restituita dallo *hit-test* per la creazione di una *Anchor* a cui legare tale contenuto 3D. Per effettuare questo ancoraggio in maniera ottimale è però necessario che il raggio utilizzato dallo *hit-testing* si vada ad intersecare con un oggetto o una superficie riconosciuti da *ARCore*.

A tal scopo, *ARCore* definisce i piani ed i punti riconosciuti come **Trackable**[13], ovvero un particolare tipo di oggetto riconosciuto che può essere tracciato nel corso del tempo. Il vantaggio principale di un *Trackable* è la possibilità di legarci delle *Anchor*, in modo tale che con lo scorrere del tempo e con l'evoluzione della scena il contenuto virtuale sia posizionato in modo preciso ed accurato. In Figura 2.5 è possibile vedere la superficie di un tavolo

[12] ARCore: Anchor, `https://developers.google.com/ar/reference/java/com/google/ar/core/Anchor`.

[13] ARCore: Trackable, `https://developers.google.com/ar/reference/java/com/google/ar/core/Trackable`.

riconosciuta come *Trackable* ed un contenuto virtuale 3D posizionato sopra ad esso tramite un'apposita *Anchor*.

Figura 2.5: Esempio di superficie riconosciuta come *Trackable* e di contenuto virtuale 3D posizionato sopra ad essa tramite una *Anchor* [67].

Dato che *ARCore* riconosce sia superfici orizzontali che verticali, i contenuti virtuali possono essere anche creati ed ancorati ad una parte di superficie angolata. Per fare ciò *ARCore* introduce i cosiddetti **Oriented Points**, dei particolari *feature points* che verranno utilizzati per stimare l'angolo della superficie riconosciuta in modo tale da poter posizionare l'oggetto tenendo in considerazione quel particolare angolo.

2.5.6 Sceneform APIs

Sceneform fornisce una serie di APIs che rendono lo sviluppo di un'applicazione in AR molto più veloce e semplice, ad esempio fornisce la classe **ArFragment**[14] che gestisce automaticamente tutte le configurazioni necessarie per il corretto funzionamento di una sessione di *ARCore*, tra cui i controlli

[14]Sceneform APIs: ArFragment, `https://developers.google.com/ar/reference/java/com/google/ar/sceneform/ux/ArFragment`.

necessari per capire se all'applicazione sono stati dati tutti i permessi oppure se la versione installata di *ARCore* è compatibile con quella richiesta.

Oltre a tutti questi controlli, *ArFragment* si occupa anche della creazione di una surface view, chiamata **ArSceneView**[15], che si integra con *ARCore* per creare la scena in AR e renderizzarla. L'*ArSceneView* è collegata ad una particolare **Scene**[16], la quale è una scena realizzata come una struttura ad albero che contiene un nodo per ogni contenuto virtuale che deve essere renderizzato. Questo significa che per creare un nuovo contenuto virtuale possiamo semplicemente creare un oggetto di tipo **Node**[17] ed assegnargli tutte le informazioni necessarie, come ad esempio che cosa deve renderizzare, qual è la sua *Anchor* oppure quali sono le azioni assegnate per permettere all'utente di potervi interagire.

Essendo la scena realizzata come una struttura ad albero, chiamata *Scene Graph*, è anche possibile assegnare un *Node* ad un altro *Node*, in modo tale da formare una sorta di relazione padre-figlio tra i contenuti virtuali, funzionalità che torna utile se ad esempio si vuole fare in modo che due contenuti virtuali si muovano e si spostino assieme mantenendo certe proprietà nella scena. Una cosa che è importante da ricordare è che un *Node* può avere più di un figlio, ma avrà sempre e solo un padre.

Per definire quali sono i modelli 3D e 2D da utilizzare per il rendering di un *Node* possiamo sfruttare *Sceneform* per creare dei **Renderable**[18], se si tratta di modelli 3D, o dei **ViewRenderable**[19], se invece si tratta di modelli 2D. La creazione di modelli composti da forme e materiali basilari può essere

[15]Sceneform APIs: ArSceneView, `https://developers.google.com/ar/reference/java/com/google/ar/sceneform/ArSceneView`.

[16]Sceneform APIs: Scene, `https://developers.google.com/ar/reference/java/com/google/ar/sceneform/Scene`.

[17]Sceneform APIs: Node, `https://developers.google.com/ar/reference/java/com/google/ar/sceneform/Node`.

[18]Sceneform APIs: Renderable, `https://developers.google.com/ar/reference/java/com/google/ar/sceneform/rendering/Renderable`.

[19]Sceneform APIs: ViewRenderable, `https://developers.google.com/ar/reference/java/com/google/ar/sceneform/rendering/ViewRenderable`.

fatta anche a livello di codice, mentre i modelli più complessi possono essere comodamente importati ed utilizzati all'interno del progetto.

In Figura 2.6 è possibile vedere un esempio di scena in AR creata tramite *ARCore* e *Sceneform* nella quale vediamo una riproduzione in Realtà Aumentata del nostro sistema solare. In questa scena i pianeti ed il Sole sono creati tramite modelli 3D importati come *Renderable*, mentre la finestra posizionata sopra è un widget standard di Android utilizzato come modello 2D per un *ViewRenderable*.

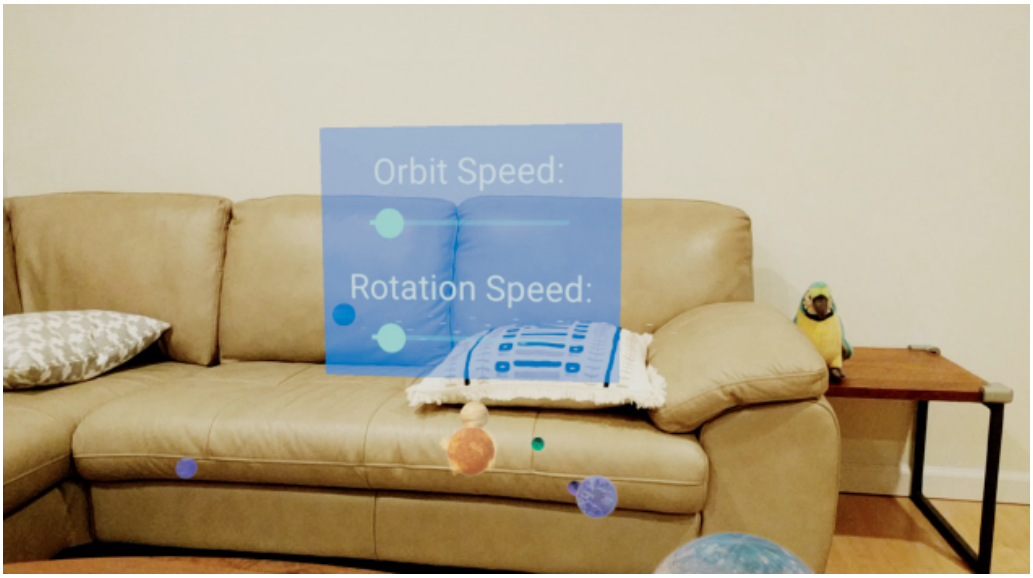

Figura 2.6: Riproduzione del sistema solare realizzata in Realtà Aumentata tramite *ARCore* e *Sceneform* [67].

Tutte le funzionalità e le operazioni legate al rendering vengono fatte tramite **Filament**[20], un motore grafico sviluppato sempre da Google per il rendering in real-time di contenuti 3D basati su modelli fisici. Ad ogni frame, *Sceneform* sfrutterà quindi questo motore grafico per creare la scena e tutti i suoi contenuti scorrendo la struttura ad albero *Scene Graph* a partire dal punto di vista della fotocamera del device.

[20]Google Filament, `https://github.com/google/filament`.

2.5.7 Altre funzionalità

Oltre alle già sopracitate funzionalità, *ARCore* ne fornisce altre che non sono state utilizzate nel corso di questa tesi, ma che è giusto menzionare. Tra queste funzionalità troviamo:

- **Augmented Images**[21]: utilizzate per il riconoscimento di immagini 2D su cui poter visualizzare dei contenuti in AR nel momento in cui vengono inquadrate e riconosciute dalla fotocamera del device;
- **Cloud Anchors APIs**[22]: sfruttate per la creazione di applicazioni in AR in ambienti collaborativi o multi-utente in cui i *feature points* e le *Anchors* vengono condivise tra diversi device al fine di operare sulla stessa scena in Realtà Aumentata contemporaneamente; in Figura 2.7 è possibile vedere un esempio in cui vengono utilizzate queste APIs per la creazione dell'app *Just a Line* [68] che permette di disegnare linee in AR in un ambiente di disegno condiviso tra più utenti.

Figura 2.7: Esempio di utilizzo delle *Cloud Anchors API* di *ARCore* [68].

[21] ARCore: Augmented Images, `https://developers.google.com/ar/reference/java/com/google/ar/core/AugmentedImage`.

[22] ARCore: Cloud Anchors APIs, `https://developers.google.com/ar/develop/java/cloud-anchors/overview-android`.

Capitolo 3

Librerie sviluppate

3.1 Progettazione della soluzione

Lo sviluppo di un'applicazione mobile in AR *location-based* necessita di 3 principali funzionalità:

1. ricavare l'orientazione del device e tutti i dati relativi ai movimenti dell'utente nello spazio che lo circonda;
2. ricavare la posizione GPS del device;
3. creare i contenuti virtuali in Realtà Aumentata.

Per sopperire a questo bisogno sono state realizzate 3 diverse librerie Android, ognuna delle quali può essere utilizzata in modo indipendente:

1. **Device Orientation**, che fornisce tutti i dati relativi all'orientazione del device ricavati direttamente dai sensori al suo interno;
2. **Device Position**, tramite la quale è possibile reperire tutte le informazioni necessarie sulla posizione GPS del device utilizzando le *Google APIs*;
3. **AR POI Experiences**, che invece si occupa della creazione dei contenuti in AR e di tutti i calcoli necessari al corretto rendering della scena sfruttando *ARCore* e *Sceneform*.

La libreria *AR POI Experiences* è il prodotto principale di questa tesi ed è stata realizzata come proposta di soluzione al problema analizzato, mentre *Device Orientation* e *Device Position* sono state realizzate per ottenere tutte le informazioni necessarie per lo sviluppo di tale soluzione. La scelta di realizzare queste due funzionalità come librerie Android indipendenti dalla soluzione proposta deriva dal fatto che non si vuole vincolare lo sviluppatore forzandolo, ad esempio, ad utilizzare le *Google APIs* per ottenere i dati geolocalizzati sulla posizione dell'utente. In questo modo lo sviluppatore che vorrà utilizzare *AR POI Experiences* potrà scegliere se sfruttare le librerie secondarie fornite oppure se ricavare tutti i dati necessari in modo autonomo ed arbitrario.

Tutte e 3 le librerie sono state sviluppate per le *API 24* di Android, ovvero utilizzando **Nougat** come versione minima di sistema operativo Android. Questa scelta è stata fatta in quanto la Realtà Aumentata, in particolare *ARCore*, è supportata solamente su device molto recenti, i quali dispongono tutti dell'ultima versione di Android. Inoltre, secondo i dati raccolti da Android e consultabili nell'apposita sezione *Distribution Dashboard* [69] sul loro sito, al 26 Ottobre 2018 la percentuale di utenti dotati di smartphone che utilizzano una versione uguale o maggiore delle *API 24* sono pari al 49.7%, percentuale che ci è sembrata più che ragionevole (Figura 3.1).

Nel corso di questo capitolo descriveremo passo dopo passo come queste librerie sono state sviluppate, le scelte implementative realizzate, le funzionalità che queste librerie offrono all'utente e come sono state integrate al fine di proporre una soluzione al problema analizzato.

3.2 Device Orientation

Come accennato in precedenza, questa libreria permette di ricavare tutti i dati necessari sull'orientazione del device utilizzandone i relativi sensori.

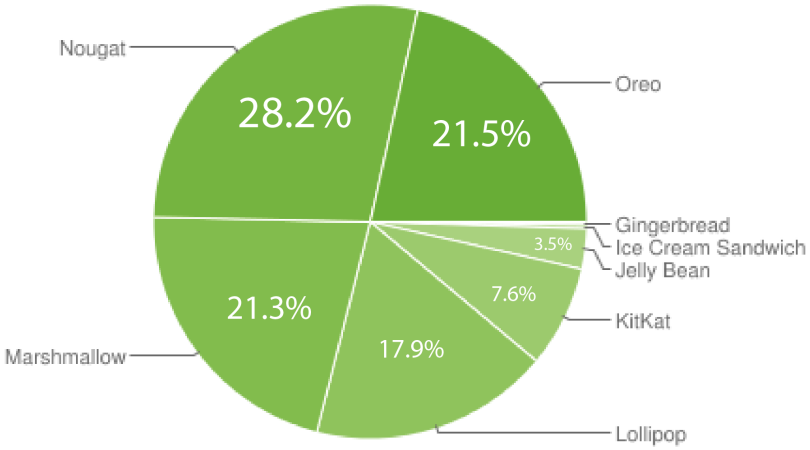

Figura 3.1: Percentuale di device che utilizzano un determinato sistema operativo di Android, aggiornato al 26 Ottobre 2018 [69].

Per prima cosa abbiamo configurato il *Manifest*[1] della libreria, file che accompagna ogni progetto Android e che si occupa di alcuni step fondamentali per il corretto funzionamento del progetto, come ad esempio descriverne le componenti, dichiarare i permessi necessari ed il livello minimo di API Android oppure descrivere quali sono le librerie necessarie all'app per il corretto funzionamento. Essendo questa una libreria Android e non un'app, nel *Manifest* andremo solamente a specificare quali sono i sensori necessari, ovvero l'accelerometro, il giroscopio ed il magnetometro.

```
1 <uses-feature
2   android:name="android.hardware.sensor.accelerometer"
3   android:required="true" />
4 <uses-feature
5   android:name="android.hardware.sensor.gyroscope"
6   android:required="true" />
7 <uses-feature
8   android:name="android.hardware.sensor.compass"
9   android:required="true" />
```

[1]Android App Manifest, `https://developer.android.com/guide/topics/manifest/manifest-intro`.

La libreria è composta da due classi:

- **DeviceOrientation**, la classe principale che contiene tutte le funzionalità offerte;
- **LowPassFilter**, che contiene i metodi per applicare un filtro passa-basso ai dati ricevuti dai sensori al fine di eliminare il rumore.

Prima di entrare nel dettaglio sulle funzionalità offerte da queste due classi, è importante definire in che modo è possibile interagire con i sensori del device per poter ricavare tutti i dati di cui abbiamo bisogno. In Android l'utilizzo dei sensori passa attraverso l'apposito *Sensor Framework*[2], il quale ti permette di gestirli grazie all'implementazione delle seguenti interfacce:

- *SensorManager*, utilizzata per creare un'istanza di un servizio tramite il quale poter interagire con i sensori del device;
- *Sensor*, che ti permette di creare un'istanza di uno specifico sensore;
- *SensorEvent*, sfruttata dal sistema per rendere accessibili le informazioni legate ad un evento, ad esempio la ricezione di nuovi dati;
- *SensorEventListener*, interfaccia utilizzata per ricevere una notifica legata ad un *SensorEvent*.

La classe **DeviceOrientation** implementa l'interfaccia *SensorEventListener*, in modo tale da poter utilizzare le due callbacks *onSensorChanged* e *onAccuracyChanged* per, rispettivamente, gestire tutti i dati grezzi ricevuti dai sensori e definire le operazioni da fare nel caso in cui venga modificata l'accuratezza di un sensore. Per gli scopi di questa tesi non è stato necessario implementare correttamente delle particolari politiche all'interno della callback *onAccuracyChanged*, ma si è lavorato solamente sulla gestione dei dati

[2]Android: Sensor Framework, `https://developer.android.com/guide/topics/sensors/sensors_overview`.

grezzi ricevuti dai sensori. In particolare, all'interno della callback *onSensorChanged* andiamo innanzitutto a memorizzare i nuovi dati ricevuti applicandogli prima il filtro passa-basso contenuto nella classe *LowPassFilter*, in modo tale da ridurre il rumore a cui i sensori sono soggetti.

L'algoritmo implementato per l'applicazione del filtro passa-basso è molto semplice, prende in input il vettore che contiene i nuovi dati, chiamato *newValues*, quello che contiene i vecchi dati, chiamato *old*, e la variabile *alpha* che determina il grado di smoothing dei dati per poi restituire in output solamente il nuovo vettore con i dati filtrati. Il vettore restituito in output viene calcolato in modo molto semplice come combinazione tra i vecchi ed i nuovi dati, in modo tale da attenuare eventuali picchi di rumore che potrebbero essersi registrati in un certo lasso di tempo.

```
1  float[] filter(float[] newValues, float[] old, float alpha) {
2    /* Se e' la prima volta che registriamo dati, allora non
3     * possiamo applicare nessun filtro.
4     */
5    if (old == null)
6      return newValues;
7    /* Applichiamo lo smoothing ai dati grezzi. */
8    for (int i = 0; i < newValues.length; i++)
9      old[i] = old[i] + alpha * (newValues[i] - old[i]);
10   /* Restituiamo in output i dati filtrati. */
11   return old;
12 }
```

A seguito di alcuni test, il valore di *alpha* ideale che, per gli scopi di questa tesi, ci ha permesso di migliorare i dati ricevuti dai sensori è pari a 0.15.

Dopo aver applicato il filtro passa-basso ai dati ricevuti all'interno della callback *onSensorChanged*, possiamo utilizzare questi dati filtrati per calcolare le 4 informazioni principali fornite da questa libreria:

- **Azimuth**, angolo lungo l'asse z del device;
- **Pitch**, angolo lungo l'asse x (lato corto) del device;
- **Roll**, angolo lungo l'asse y (lato lungo) del device;

- **Bearing**, angolo con il Nord magnetico misurato in senso orario.

Tutti e 4 gli angoli sono calcolati in gradi decimali, per maggiori informazioni fare riferimento all'Appendice A.

Come è possibile vedere graficamente in Figura 3.2[3] l'*Azimuth* ed il *Bearing* rappresentano entrambi lo stesso angolo, la loro unica differenza risiede nel range in cui sono calcolati: il primo è calcolato nel range $[-180.0, +180.0]$ mentre il secondo in quello $[0.0, 360.0]$. Il parametro *Bearing* viene definito solamente per pura comodità, come vedremo nel corso dei prossimi capitoli.

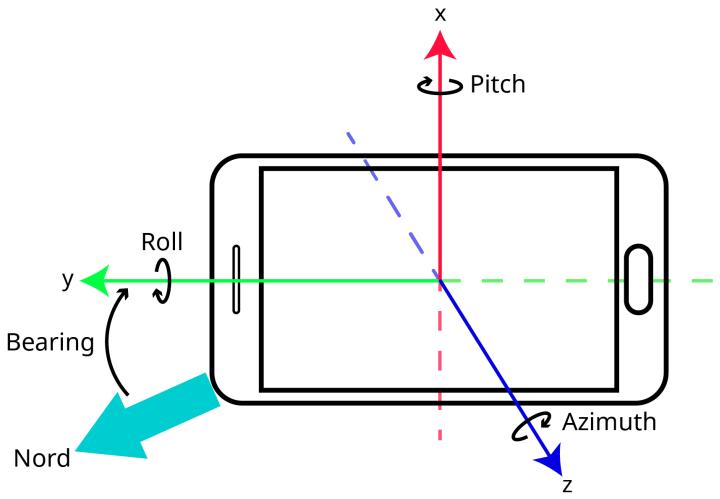

Figura 3.2: Rappresentazione grafica degli assi utilizzati dai sensori e degli angoli *Azimuth*, *Pitch*, *Roll* e *Bearing*.

Seguendo le norme e le convenzioni fornite dalla documentazione Android, le operazioni legate al calcolo di questi angoli sono state fatte tutte in background sfruttando un *AsyncTask*[4], in modo tale da non bloccare lo

[3]Icona del device realizzata da *Freepik*, `https://www.freepik.com/`. Scaricata dalla piattaforma *FlatIcon*, `www.flaticon.com`.

[4]Android AsyncTask, `https://developer.android.com/reference/android/os/AsyncTask`.

UI thread[5] principale. All'interno dell'*AsyncTask* abbiamo poi effettuato le seguenti operazioni:

1. ricavare la *Rotation Matrix*[6] utilizzando i dati ricevuti da accelerometro e magnetometro;
2. se abilitato, rimappare[7] il sistema di coordinate utilizzato dal *SensorManager* secondo le indicazioni date dall'utente;
3. ricavare l'*Orientation Vector*[8] utilizzando la *Rotation Matrix* calcolata precedentemente;
4. ricavare *Azimuth*, *Pitch* e *Roll* dall'*Orientation Vector* e convertirli da radianti a gradi;
5. calcolare il *Bearing* utilizzando l'*Azimuth* in gradi.

La *Rotation Matrix* ricavata allo step 1 è una matrice che ci fornisce le informazioni sul sistema di coordinate mondo, fondamentali per poter calcolare l'*Orientation Vector* allo step 3, ovvero il vettore che contiene le informazioni su come è orientato il device grazie al quale possiamo ricavare i valori dei tre angoli *Azimuth*, *Pitch* e *Roll*.

Lo step 2 è necessario per far sì che il sistema di coordinate utilizzato dai sensori Android combaci con quello utilizzato da *ARCore* per l'ambiente in Realtà Aumentata. Dato che *ARCore* è una libreria basata su OpenGL,

[5]In Android lo *UI thread* è il thread principale che si occupa dell'esecuzione di una applicazione, in particolare si occupa della creazione e della gestione di tutti i componenti (*Activity*, *Service*, ...).

[6]Android Sensor Manager: getRotationMatrix, `https://developer.android.com/reference/android/hardware/SensorManager#getRotationMatrix(float[],%20float[],%20float[],%20float[])`.

[7]Android Sensor Manager: remapCoordinateSystem, `https://developer.android.com/reference/android/hardware/SensorManager.html#remapCoordinateSystem(float[],%20int,%20int,%20float[])`.

[8]Android Sensor Manager: getOrientation, `https://developer.android.com/reference/android/hardware/SensorManager#getOrientation(float[],%20float[])`.

il sistema di coordinate utilizzato per l'AR è proprio quello destrorso di OpenGL. In particolare, il sistema di coordinate utilizzato dai sensori fa riferimento al device e l'asse y sarà sempre posizionato lungo il *lato lungo* del device, mentre il sistema di coordinate utilizzato da OpenGL fa riferimento all'orientazione corrente del device, quindi l'asse y sarà sempre orientato verso l'*alto*.

Come è possibile vedere in Figura 3.3, quando il dispositivo si trova in modalità *portrait* i due sistemi di riferimento combaciano, mentre quando si passa in modalità *landscape* non combaciano più. A causa di ciò, i dati ricevuti dai sensori quando il device si trova in modalità *landscape* non sono consistenti con il sistema di coordinate in AR, quindi è necessario rimappare il sistema di coordinate (step 2) per farli di nuovo combaciare.

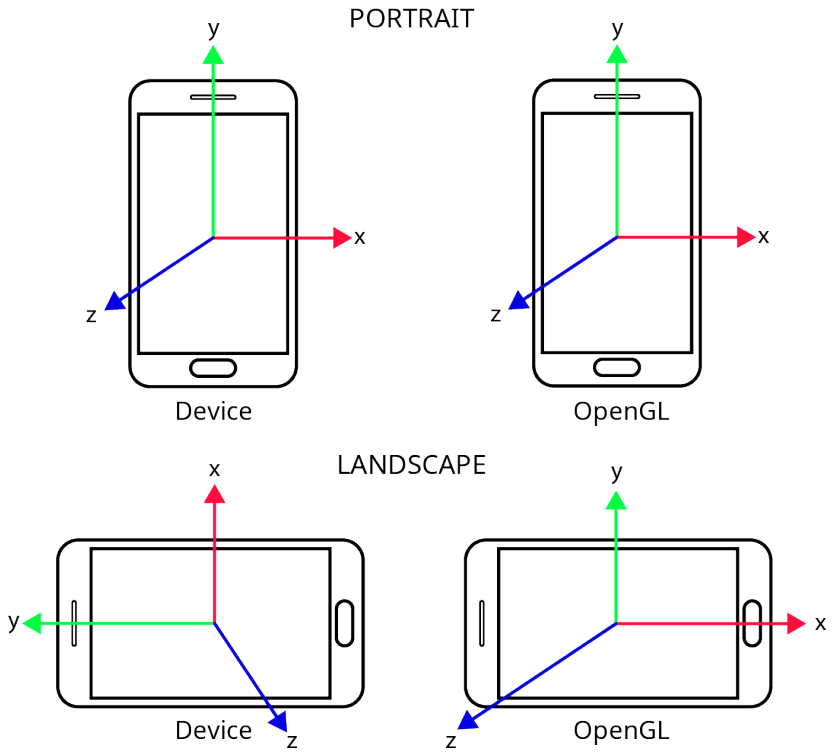

Figura 3.3: Differenze tra il sistema di coordinate utilizzato dal device e quello utilizzato da OpenGL.

Altre due funzionalità offerte da questa libreria sono la registrazione e la

rimozione dei *listener* utilizzati per la ricezione delle informazioni dai sensori.

3.3 Device Position

La seconda informazione necessaria per lo sviluppo di app in AR *location-based* è la posizione GPS dell'utente. Per ottenerla è stata sviluppata questa libreria sfruttando le *Google APIs*, nello specifico utilizzando un **Fused Location Provider Client**[9] che combina differenti segnali ricevuti da più sensori per ricavare tutte le informazioni sulla posizione che l'app richiede. Il *Fused Location Provider* fornisce una serie di semplici APIs per gestire tutte le diverse tecnologie utili alla localizzazione, come il GPS ed il Wi-Fi, e per specificare il livello desiderato di accuratezza del dato. Permette inoltre di ricavare l'ultima posizione GPS conosciuta dell'utente da altre app presenti nel device, evitando così di ripetere operazioni non necessarie al fine di ottimizzare e ridurre il consumo della batteria.

Per prima cosa, dato che vogliamo utilizzare le *Google APIs* per Android, è necessario impostare il progetto importando l'SDK nel file di configurazione di *Gradle*[10]. Quando compiliamo un progetto Android, tutto il codice sorgente e le risorse utilizzate dall'app vengono compilate per generare uno o più APK, in modo tale che sia possibile testare, rilasciare e distribuire al pubblico l'app. Per effettuare questo processo *Android Studio* utilizza appunto *Gradle*, una serie di tools che permettono di gestire tutto il processo in maniera automatica e flessibile leggendo tutte le configurazioni necessarie all'interno di appositi file chiamati *build.gradle*. Per far sì che la nostra libreria possa sfruttare le *Google APIs* è quindi necessario aggiungerle come dipendenza all'interno del nostro file *Gradle* specificando quali sono le APIs di cui abbiamo bisogno e qual è la versione che vogliamo utilizzare. Nel no-

[9]Google APIs for Android: FusedLocationProviderClient, `https://developers.google.com/android/reference/com/google/android/gms/location/FusedLocationProviderClient`.

[10]Gradle Build Tool, `https://gradle.org/`.

stro caso, le APIs importate sono le *Google Play Services Location APIs*[11] versione 16.0.0.

Successivamente andiamo a specificare all'interno del *Manifest* quali sono i permessi di cui la nostra libreria ha bisogno, ovvero i permessi relativi alle funzionalità di geolocalizzazione del device sia tramite GPS che tramite Wi-Fi.

```
<uses-permission
  android:name="android.permission.ACCESS_FINE_LOCATION" />
<uses-permission
  android:name="android.permission.ACCESS_COARSE_LOCATION" />
<uses-feature
  android:name="android.hardware.location.gps"
  android:required="true" />
<uses-feature
  android:name="android.hardware.location.network"
  android:required="true" />
```

Una volta configurati i file principali che definiscono quali sono le APIs utilizzate e quali sono i permessi necessari per il corretto funzionamento di questa libreria, passiamo alla vera e propria creazione di *Device Position*. La libreria si compone di quattro classi:

- **DevicePosition**, la classe che fornisce le funzionalità principali per ottenere la posizione GPS dell'utente e tutte le informazioni necessarie;
- **PositionUtils**, fornisce metodi utili per determinare se la nuova posizione GPS è più accurata o più recente della precedente;
- **FetchAddressTask**, un *AsyncTask* che processa in background le informazioni ricevute sulla posizione GPS per ricavarne un indirizzo associato;
- **FetchAddressResult**, un modello utilizzato per restituire allo *UI thread* il risultato calcolato dal *FetchAddressTask*.

[11] Google APIs: Play Services Location, `https://developers.google.com/android/reference/com/google/android/gms/location/package-summary`.

Quando viene istanziato un oggetto della classe *DevicePosition* la prima cosa che viene fatta è quella di inizializzare il *location tracker*, ovvero viene inizializzato il *Fused Location Provider Client* in modo tale che sia possibile interagire con le tecnologie all'interno del device per ottenerne la posizione GPS attuale.

Una volta inizializzato il *Fused Location Provider Client* è necessario richiedere gli aggiornamenti sulla posizione del device creando un'apposita **LocationRequest**[12], ovvero un oggetto che contiene le informazioni sulla qualità del servizio richiesto al *Fused Location Provider*. In particolare, all'intero della *LocationRequest* specifichiamo qual è la frequenza con cui vogliamo ricevere gli aggiornamenti sulla posizione, l'accuratezza del dato e la latenza, intesa come intervallo massimo di tempo che possiamo aspettare prima di ricevere un *batch* contenente uno o più aggiornamenti sulla posizione del device.

Come parametro per definire l'accuratezza del nostro dato abbiamo impostato *PRIORITY_HIGH_ACCURACY*, valore che ci permette di ricevere aggiornamenti sulla posizione con la maggiore accuratezza possibile a discapito di un maggiore consumo di batteria dovuto all'interrogazione di tutte le tecnologie presenti nel device. Per gli scopi di questa tesi, è stato un *trade-off* necessario al fine di migliorare la qualità dell'esperienza offerta all'utente. Quando invece specifichiamo la frequenza utilizziamo due metodi: il primo, *setInterval(long)*, ci permette di indicare qual è il tasso con cui vorremmo ricevere gli aggiornamenti, mentre il secondo, *setFastestInterval(long)* ci permette di impostare un intervallo esatto di tempo, in modo tale che la nostra libreria non possa ricevere aggiornamenti sulla posizione più velocemente del valore che abbiamo impostato qui.

La *LocationRequest* creata e configurata verrà poi passata al *Fused Location Provider Client* assieme ad una **LocationCallback**[13] al fine di poter

[12]Google APIs: LocationRequest: `https://developers.google.com/android/reference/com/google/android/gms/location/LocationRequest`.

[13]Google APIs: LocationCallback, `https://developers.google.com/android/reference/com/google/android/gms/location/LocationCallback`.

ricevere gli aggiornamenti sulla posizione GPS del device. La *LocationCallback* è una funzione creata dall'utente e passata alla libreria *Device Position* in cui vengono specificate le operazioni da eseguire sul dato ogni volta che si riceve una nuova posizione GPS. Dato che la libreria ha come target tutti i sistemi operativi Android dalle *API 24* in poi, dobbiamo controllare a *runtime* se disponiamo di tutti i permessi necessari al corretto utilizzo della libreria e delle *Google APIs* e, se necessario, richiedere i permessi mancanti all'utente. Tutte le funzionalità elencate e descritte finora permettono una corretta configurazione del *Fused Location Provider Client* al fine di ricevere aggiornamenti sulla posizione GPS del device.

Una volta effettuate tutte le configurazioni necessarie, è possibile sfruttare le seguenti funzionalità:

1. rimuovere gli aggiornamenti sulla posizione GPS;

2. ottenere la posizione GPS attuale del device;

3. controllare se l'ultimo aggiornamento ricevuto contiene una posizione GPS migliore della precedente.

Per effettuare il controllo descritto al punto 3 sfruttiamo la classe *PositionUtils* introdotta precedentemente, la quale offre il metodo *isBetterLocation(Location, Location)*. Questo metodo prende in input due oggetti di tipo *Location*[14], ovvero due coordinate GPS, e restituisce in output un valore booleano: *true* nel caso in cui la posizione GPS passata come primo parametro sia migliore, in termini di accuratezza e tempo, rispetto alla posizione GPS passata come secondo parametro, altrimenti restituisce *false*. Nel nostro caso, all'interno della classe *DevicePosition*, questo metodo viene utilizzato per confrontare la posizione o le posizioni GPS ricevute dall'ultimo aggiornamento dal *Fused Location Provider* con la posizione GPS attualmente memorizzata. In particolare, la posizione passata come primo parametro è

[14]Android APIs: Location, `https://developer.android.com/reference/android/location/Location`.

quella ricevuta dall'ultimo aggiornamento (*newLocation*), mentre il secondo parametro è la migliore posizione GPS attualmente memorizzata ed utilizzata (*currentLocation*). Di seguito riportiamo il codice utilizzato dal metodo con la spiegazione tramite commenti degli step realizzati.

```
1  public boolean isBetterLocation(
2    Location newLocation, Location currentLocation) {
3    /* La nuova posizione e' sicuramente la migliore. */
4    if (currentLocation == null)
5      return true;
6    /* Controlliamo se la nuova posizione GPS e' piu' recente
7     * di quella memorizzata.
8     */
9    long t = newLocation.getTime() - currentLocation.getTime();
10   /* Se la nuova posizione e' significativamente piu' recente
11    * allora e' la migliore, se invece e' significativamente
12    * meno recente, allora quella attualmente memorizzata e'
13    * la migliore.
14    */
15   if (t > update_time)
16     return true;
17   else if (t < -update_time)
18     return false;
19   /* Nel caso in cui non ci siano grosse differenze
20    * temporali, controlliamo l'accuratezza della nuova
21    * posizione GPS rispetto a quella memorizzata.
22    */
23   int a = (int) (newLocation.getAccuracy() -
24       currentLocation.getAccuracy());
25   /* Restituiamo true nel caso in cui la nuova posizione GPS
26    * sia molto piu' accurata di quella memorizzata oppure
27    * nel caso in cui sia piu' recente ma non drasticamente
28    * meno accurata, altrimenti false.
29    */
30   return a < 0 || (t > 0 && a < 200);
31 }
```

La variabile *update_time* utilizzata all'interno del metodo è un parametro

della classe *PositionUtils* che viene inizializzato di default pari a 60 secondi, ma l'utente può selezionare una durata di tempo arbitraria in base alle sue preferenze e necessità. Sfruttando il metodo appena descritto, è possibile utilizzare il metodo *checkAndSetCurrentBestLocation(LocationResult)* all'interno della classe *DevicePosition* per controllare se la nuova o le nuove posizioni GPS ricevute sono migliori di quella attualmente memorizzata ed, in caso affermativo, aggiornare la posizione GPS memorizzata con la migliore tra quelle ricevute.

Si è scelto di dare maggiore importanza ad una posizione GPS più recente rispetto ad una più accurata in modo tale che un'eventuale app mobile realizzata sfruttando questa libreria possa catturare facilmente gli spostamenti dell'utente nel tempo. Inoltre, per gli scopi di questa tesi, avremo bisogno di poter lavorare con un posizione temporalmente recente ed aggiornata poiché lo riteniamo un fattore chiave per la creazione di esperienze in AR legate a punti di interesse.

Per quanto riguarda invece la classe *FetchAddressTask*, essa non viene utilizzata direttamente all'interno della classe *DevicePosition* ma è una funzionalità aggiuntiva offerta che l'utente può sfruttare o meno. La classe, come anticipato in precedenza, è un *AsyncTask* che effettua in background tutti i calcoli necessari per trasformare una coordinata GPS in un indirizzo. Al suo interno si trova anche un'interfaccia chiamata *OnFetchAddressTaskCompleted*, implementabile all'interno di una *Activity*, il cui metodo *onFetchAddressTaskCompleted* viene invocato ogni volta che l'*AsyncTask* termina i calcoli ed ottiene un risultato di tipo *FetchAddressResult*, il quale contiene un campo **Address**[15] con l'indirizzo calcolato a partire dalla coordinata GPS ed una serie di altri campi aggiuntivi, come la latitudine, la longitudine, l'altitudine, l'accuratezza del risultato ed un eventuale messaggio di errore nel caso in cui qualcosa sia andato storto.

All'interno dell'*AsyncTask* i calcoli che vengono fatti per trasformare una

[15] Android APIs: Address, `https://developer.android.com/reference/android/location/Address`.

coordinata GPS in un indirizzo sfruttano le funzionalità della classe **Geocoder**[16] di Android, la quale gestisce tutte le operazioni di *geocoding* e di *geocoding inverso*. Quando parliamo di *geocoding* intendiamo il processo di trasformare un indirizzo in una coordinata GPS composta da latitudine e longitudine, mentre il *geocoding inverso* è esattamente il processo che abbiamo utilizzato noi, ovvero la trasformazione di una coordinata GPS in un indirizzo.

3.4 AR POI Experiences

Al fine di offrire una possibile soluzione al problema analizzato utilizzando una tecnologia emergente e sempre più popolare come la Realtà Aumentata, abbiamo sviluppato la libreria Android *AR POI Experiences*. Come accennato precedentemente, questa libreria sfrutta *ARCore* e *Sceneform* per la creazione di contenuti in Realtà Aumentata e, per poter creare esperienze legate a punti di interesse, ha bisogno di ricevere le informazioni sulla posizione GPS del device e sulla sua attuale orientazione, informazioni che vengono fornite dalle librerie *Device Orientation* e *Device Position* appena descritte.

Grazie alla libreria Android *AR POI Experiences* è possibile sia creare dei contenuti in Realtà Aumentata posizionabili in corrispondenza di *POIs*, che creare una scena che può essere vista come un contenitore configurabile dall'utente tramite le seguenti operazioni:

- aggiunta o rimozione di contenuti in AR;
- gestione di eventi associati ai contenuti in AR;
- gestione dell'aggiornamento della scena;
- raggruppamento di contenuti in AR sovrapposti all'interno della scena;

[16] Android APIs: Geocoder, `https://developer.android.com/reference/android/location/Geocoder`.

- clipping di contenuti in AR all'interno della scena.

Per quanto riguarda invece i contenuti in AR è possibile definire quale modello 2D o 3D utilizzare per il rendering, associare eventi legati al tocco del contenuto da parte dell'utente oppure legati al ciclo di rendering del contenuto, specificare il *POI* nel quale posizionare il contenuto in AR oppure applicarci un'animazione.

Nel corso del prossimo capitolo descriveremo nel dettaglio come queste funzionalità sono state implementate, spiegando e motivando le varie scelte implementative che sono state adottate.

Capitolo 4

AR POI Experiences

4.1 Struttura del progetto

Come per le altre librerie descritte precedentemente, il primo step è la configurazione del *Manifest*. In questo caso al suo interno richiediamo i permessi per poter utilizzare la fotocamera del device ed esplicitiamo che la libreria utilizzerà la fotocamera per la Realtà Aumentata.

```
<uses-permission
    android:name="android.permission.CAMERA" />
<uses-feature
    android:name="android.hardware.camera.ar"
    android:required="true" />
```

Successivamente specifichiamo all'interno del file *build.gradle* le librerie da importare nel progetto, ovvero *ARCore* e *Sceneform*. Dato che entrambe sono librerie in continuo sviluppo, la versione importata è stata continuamente aggiornata di pari passo: all'inizio dei lavori la versione di *ARCore* e di *Sceneform* era la 1.2.0, mentre al momento della scrittura della tesi la versione in uso è la 1.5.0.

La libreria è composta da un totale di 8 classi e 2 packages, i quali sono a loro volta composti da 4 classi ciascuno. Di seguito riportiamo una breve

descrizione delle classi più importanti, in modo tale da fornire un po' di terminologia che verrà utilizzata nel corso di questo capitolo:

- **ArPoiScene**: classe principale della libreria che permette di creare e configurare la scena in Realtà Aumentata;
- **RealPoiNode**: rappresenta un singolo contenuto virtuale posizionato in corrispondenza di un *POI*;
- **ArPoiAnchorNode**: gestisce il ciclo di rendering del contenuto virtuale all'interno della scena in AR;
- **Cluster**: rappresenta un singolo *cluster*, inteso come raggruppamento di *RealPoiNode* sovrapposti all'interno della scena; fa parte del package *clustering* che si trova all'interno della libreria;
- **ArPoiMathUtils**: classe che fornisce i metodi utilizzati per la maggior parte dei calcoli relativi al posizionamento di un contenuto virtuale in corrispondenza di un *POI*; fa parte del package *utils* che si trova all'interno della libreria.

Ogni classe è dotata di almeno un costruttore, in cui poter inizializzare l'oggetto definendo i parametri obbligatori, ed una *inner class*[1] definita come *Builder Pattern* per semplificare la creazione e la configurazione di tutti i parametri opzionali da parte dello sviluppatore.

4.2 Configurazione dei contenuti virtuali

La creazione dei contenuti virtuali da visualizzare in corrispondenza di un *POI* avviene semplicemente istanziando l'apposita classe *RealPoiNode* citata precedentemente. I parametri obbligatori da specificare per poter creare un *RealPoiNode* sono la coordinata GPS del *POI* in cui posizionarlo (latitudine,

[1] Java: Inner Class, `https://docs.oracle.com/javase/tutorial/java/javaOO/innerclasses.html`.

longitudine ed altitudine) ed il modello 2D o 3D utilizzato per il rendering. La creazione dei modelli non è una funzionalità fornita dalla libreria poiché si voleva lasciare una maggiore libertà allo sviluppatore: per crearli esistono infatti diversi modi, tutti ugualmente corretti. In Appendice B vengono forniti maggiori dettagli su alcune delle possibili modalità di creazione.

Una volta inizializzato un *RealPoiNode* lo sviluppatore può configurarne i parametri opzionali tramite il *Builder Pattern*. In particolare, le opzioni configurabili sono le seguenti:

- aggiunta di un *ViewRenderable*, ovvero di un contenuto virtuale 2D, associato al *RealPoiNode*;
- definirne il comportamento nel caso in cui sia sovrapposto ad uno più *RealPoiNode* nella scena;
- definire l'azione da eseguire se l'utente effettua un *tap* sul *RealPoiNode* visualizzato nella scena;
- definire un particolare comportamento da eseguire ad ogni ciclo di rendering.

Come è possibile vedere in Figura 4.1, l'aggiunta di un contenuto virtuale 2D al *RealPoiNode* risulta utile quando, ad esempio, oltre al *Renderable* si vuole visualizzare un titolo o una descrizione del punto di interesse che si sta rappresentando. Inoltre, dato che *Sceneform* utilizza una struttura ad albero per effettuare il rendering della scena, aggiungere il *ViewRenderable* come nodo figlio del *Renderable* associato al *RealPoiNode* fa sì che ad ogni aggiornamento della scena entrambi si aggiornino correttamente, senza il bisogno di specificare una seconda *Anchor* per il contenuto 2D.

4.2.1 Gestione eventi

Per migliorare l'esperienza dell'utente, abbiamo fornito allo sviluppatore i mezzi per assegnare degli eventi ai *RealPoiNode*. Come accennato in precedenza, è possibile definire due tipi di eventi:

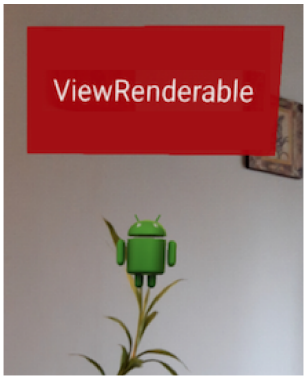

Figura 4.1: Esempio di *Renderable* e *ViewRenderable* associati ad un *POI*.

1. *onTapEvent*, ovvero l'evento da attivare se l'utente effettua un *tap* sul *Renderable*;

2. *renderEvent*, che invece è un evento da eseguire sempre ad ogni ciclo di rendering della scena.

Entrambi gli eventi sono modellati tramite un'apposita *Functional Interface*[2], un'interfaccia in cui c'è esattamente un solo metodo astratto che offre allo sviluppatore la possibilità di definire qualunque tipo di operazione inerente al *RealPoiNode*.

La principale ed unica distinzione tra i due eventi è semplicemente il momento in cui vengono attivati: nel caso dell'*onTap* sarà esattamente quando viene registrato un *tap* sul *Renderable*, utile ad esempio se si vogliono ottenere maggiori informazioni sul *POI* aprendo una nuova *Activity*, mentre nel caso del *renderEvent* l'azione verrà eseguita ad ogni *frame*. Definire un *renderEvent* diventa utile, per esempio, nel caso in cui si voglia mostrare all'interno di un *ViewRenderable* la distanza in metri tra la posizione dell'utente ed il *POI* inquadrato: ad ogni ciclo di rendering la distanza verrà calcolata ed aggiornata (Figura 4.2).

[2]Java 8: Annotation Type FunctionalInterface, `https://docs.oracle.com/javase/8/docs/api/java/lang/FunctionalInterface.html`.

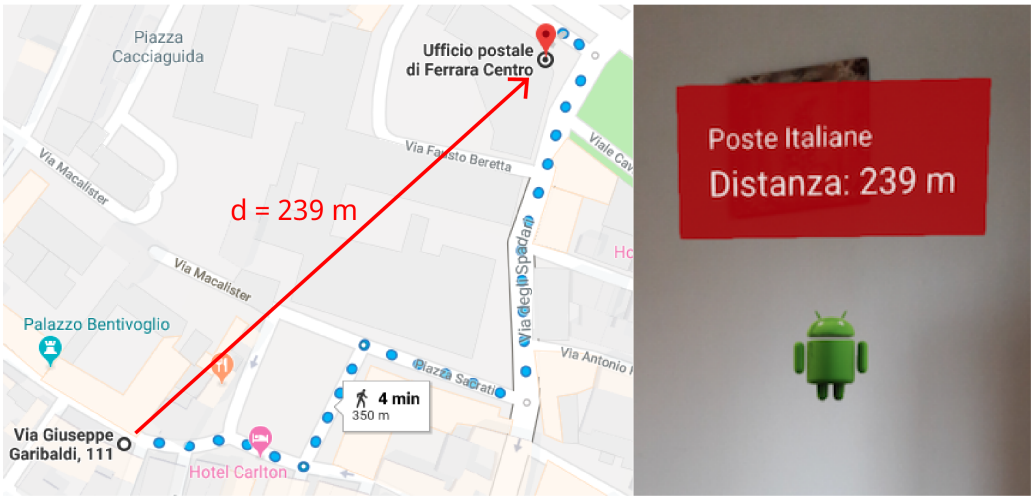

Figura 4.2: Esempio di *renderEvent* associato ad un *Renderable* in cui mostriamo la distanza in metri tra il device ed il *POI*. A sinistra vediamo su Google Maps la posizione dell'utente e quella dell'ufficio postale, mentre a destra il contenuto virtuale creato in Realtà Aumentata.

4.3 Creazione della scena

Una volta creati una serie di *RealPoiNode* è possibile aggiungerli ad una *ArPoiScene*. La creazione della scena avviene semplicemente specificando qual è l'*ArSceneView* in cui visualizzare tutti i contenuti virtuali. Tra le scelte implementative, una particolarmente importante è stata appunto come strutturare la creazione e la configurazione della scena: si è deciso di fornire la massima libertà allo sviluppatore, in modo tale che potesse configurare ogni aspetto della scena in modo arbitrario, il tutto fornendogli un modo semplice e veloce per tale configurazione, ovvero un *Builder Pattern*.

Di seguito riportiamo quali sono le opzioni configurabili per una scena:

- aggiunta o rimozione di un *RealPoiNode*;
- attivazione o disattivazione di un *Refresh Timer*, utilizzato per definire ogni quanti millisecondi la scena si deve aggiornare;

- attivazione o disattivazione del *Clipping* per rimuovere i *Renderable* troppo vicini, troppo lontani oppure esterni alla porzione di ambiente inquadrata con la fotocamera del device;
- attivazione o disattivazione del *Clustering* per raggruppare due o più *Renderable* sovrapposti sulla scena.

Una volta configurata tutta la scena con i parametri desiderati non resta che richiamare il metodo *processFrame(Frame, Location, int)*, il quale prende in input i seguenti parametri: il *frame* corrente da processare, la posizione GPS dell'utente ed il *Bearing* attuale del device.

4.4 Clipping

Quando parliamo di *Clipping* ci riferiamo al procedimento utilizzato nella *Computer Graphics* per determinare qual è la zona di interesse in cui effettuare il rendering, evitando quindi di renderizzare tutti i contenuti che cadono fuori da questa zona al fine di migliorare le performance delle funzionalità offerte dalla libreria *AR POI Experiences*. In particolare, nel nostro caso lo sviluppatore può indicare la zona di interesse abilitando o disabilitando 3 diverse tipologie di *Clipping*:

1. *Near Plane Clipping*;
2. *Far Plane Clipping*;
3. *Field of View Clipping*.

I primi due sono utilizzati, rispettivamente, per indicare quali sono la distanza minima e la distanza massima, entrambe specificate in metri, in cui renderizzare i *Renderable* associati a punti di interesse, in modo tale da non visualizzare *POIs* troppo vicini alla fotocamera del device oppure troppo distanti dall'utente. Possono essere configurati, abilitati o disabilitati in modo indipendente l'uno dall'altro, per esempio lo sviluppatore potrebbe abilitare

soltanto il *Far Plane Clipping*. In Figura 4.3[3] vediamo un esempio in cui vengono applicati entrambi i *Clipping* menzionati, per cui gli unici *POIs* che saranno renderizzati saranno quelli che si troveranno all'interno dei due limiti specificati.

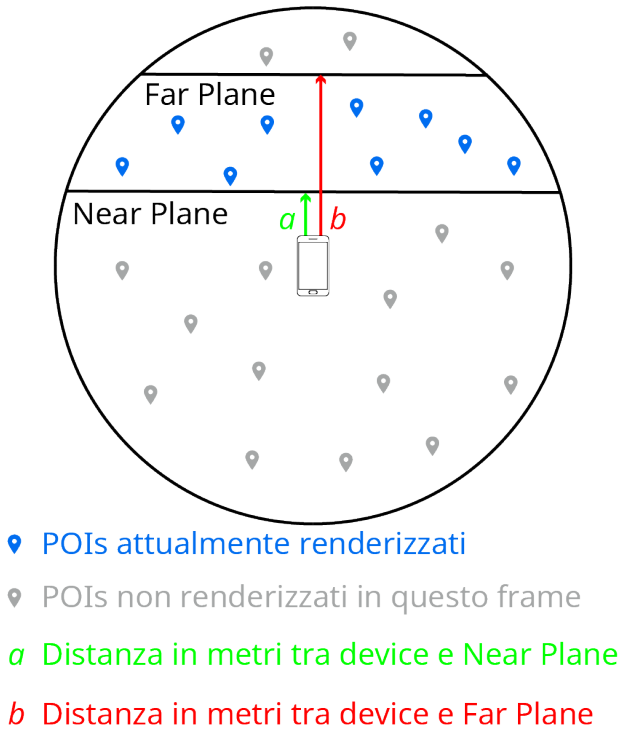

Figura 4.3: *Near Plane Clipping* e *Far Plane Clipping* applicato alla scena.

Il *Field of View Clipping*[4] invece, per comodità abbreviato in *FOV Clipping*, viene utilizzato per evitare il rendering di tutti i *Renderable* che si trovano all'esterno di un particolare range definito dallo sviluppatore. Il range va specificato in gradi e si utilizza la fotocamera come punto centrale di questo range, in modo tale che lo sviluppatore possa, per esempio, definire

[3]Icona utilizzata per indicare un *POI* creata da *Smashicons*, `https://www.flaticon.com/authors/smashicons`. Scaricata dalla piattaforma *FlatIcon*, `https://www.flaticon.com`.

[4]Il termine *Field of View* è spesso utilizzato per indicare la porzione di spazio che un utente vede in un particolare momento senza muovere gli occhi, la testa o il collo.

un range pari al cono di vista della fotocamera, ovvero pari alla porzione di ambiente che la fotocamera del device riesce ad inquadrare.

In Figura 4.4 possiamo vedere l'angolo identificato con il nome *a*, che rappresenta il range in gradi specificato dallo sviluppatore, e l'angolo identificato con il nome *b*, che sarà esattamente la metà di *a*. Utilizzando questo *Clipping* la scena effettuerà il rendering di tutti i *Renderable* che si trovano all'interno di questo range, colorati in blu, mentre tutti gli altri rimarranno memorizzati all'interno della scena ma non saranno renderizzati in questo *frame*, colorati in grigio. Come si vede in Figura 4.4, nel caso in cui un *POI* sia anche solo parzialmente all'interno del cono di vista definito, il suo *Renderable* viene comunque renderizzato interamente.

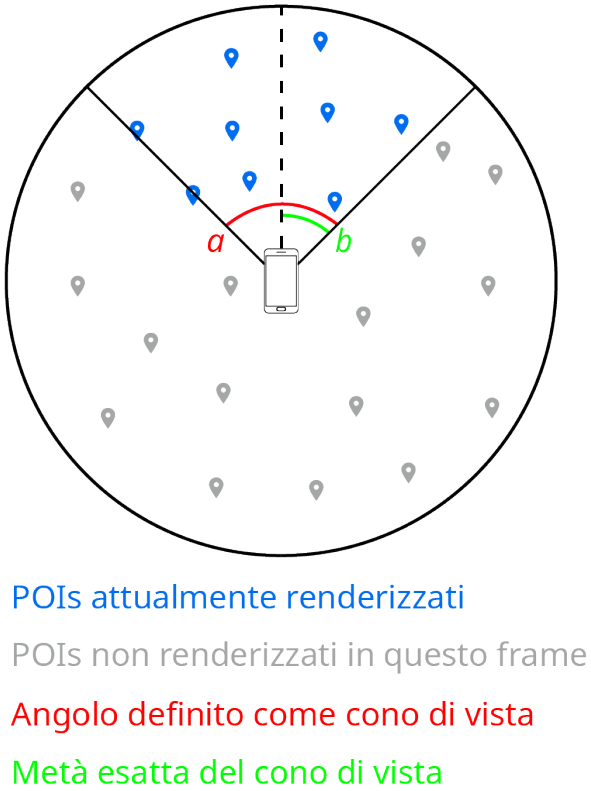

Figura 4.4: *FOV Clipping* applicato alla scena.

4.5 Aggiornamento della scena

L'aggiornamento dell'*ArPoiScene* è controllato da un parametro booleano chiamato *sceneIsUpToDate*: quando è *true* significa che la scena è già aggiornata, se invece è *false* dobbiamo aggiornarla. Tra le opzioni configurabili abbiamo un *Refresh Timer* che ogni n millisecondi, con n definito dallo sviluppatore, va a settare il parametro booleano a *false* forzando la scena ad aggiornarsi, in modo tale che tutti i *Renderable* siano sempre posizionati in modo corretto in base ai movimenti dell'utente.

Il *Refresh Timer* ed il parametro booleano appena descritti ci permettono di evitare l'aggiornamento della scena ad ogni *frame*, operazione che comporterebbe un costo computazionale non indifferente. Consideriamo ad esempio un'applicazione con un *frame rate*[5] pari a *30 FPS*[6]: se aggiornassimo la scena ad ogni *frame* dovremmo ripetere tutti i passaggi ed i calcoli circa 30 volte al secondo, sprecando così risorse per aggiornare una scena che molto probabilmente non è nemmeno cambiata nell'arco del secondo trascorso. Purtroppo però, utilizzare soltanto il timer per aggiornare la scena crea alcune problematiche importanti:

- se lo sviluppatore ha abilitato il *FOV Clipping* nella scena, verranno renderizzati soltanto i *Renderable* che si trovano nel suo cono di vista nel *frame* in cui la scena viene aggiornata; il problema nasce quando l'utente si muove nell'ambiente guardandosi attorno: non vedrà nessun altro *Renderable* finché il *Refresh Timer* non aggiornerà la scena (Figura 4.5);

- un caso simile capita se lo sviluppatore ha abilitato almeno uno tra il *Near Plane Clipping* ed il *Far Plane Clipping* nella scena; l'utente vedrà tutti i *Renderable* che si trovavano all'interno dei limiti definiti nel *frame*

[5]Con *frame rate* si intende la frequenza di riproduzione dei fotogrammi che compongono un video oppure un'animazione.

[6]L'acronimo *FPS* sta per *frame per second* ed è l'unità di misura utilizzata per esprimere il *frame rate*. Un'altra unità di misura spesso utilizzata sono gli Hertz.

in cui la scena è stata aggiornata, ma se si muove e la sua posizione GPS cambia nel tempo che intercorre tra due aggiornamenti della scena, non potrà vedere i nuovi *POIs* che entrano nei limiti aggiornati (Figura 4.6).

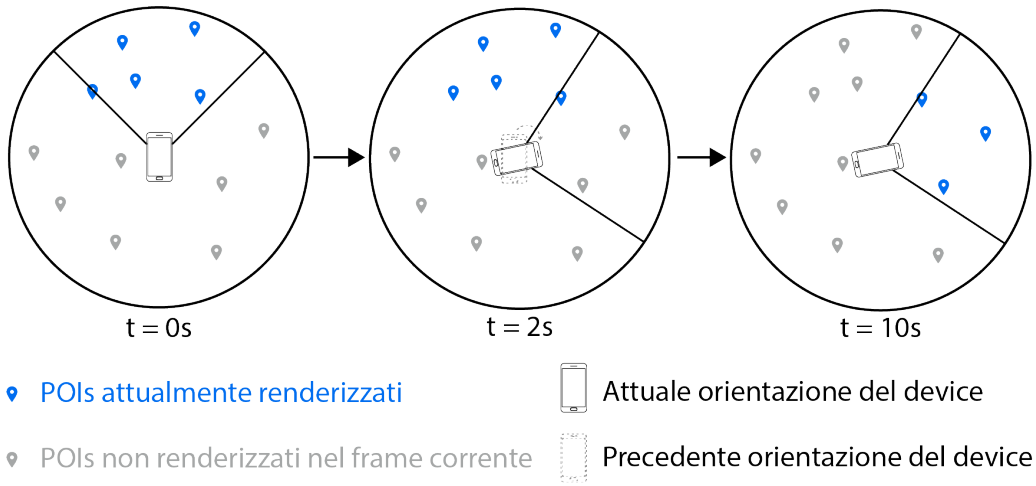

Figura 4.5: Problema di aggiornamento della scena quando il *FOV Clipping* è abilitato. Nell'esempio il *Refresh Timer* aggiorna la scena ogni 10 secondi, quindi se l'utente si muove tra due aggiornamenti non vedrà correttamente tutti i *POIs*.

Al fine di risolvere queste due problematiche che si sono presentate abbiamo sviluppato altre due modalità di aggiornamento della scena che si abilitano o disabilitano automaticamente assieme al relativo *Clipping*. In particolare, nel caso in cui venga abilitato uno tra il *Near Plane Clipping* ed il *Far Plane Clipping* verrà abilitato anche l'aggiornamento della scena in base agli spostamenti della posizione GPS dell'utente, mentre nel caso in cui venga abilitato il *FOV Clipping* verrà abilitato l'aggiornamento della scena in base ai movimenti dell'utente registrati dai sensori del device: grazie a queste due funzionalità è quindi possibile aggiornare la scena basandosi sul movimento dell'utente nell'ambiente che lo circonda, in modo tale che la scena contenga sempre i *Renderable* corretti in base alle configurazioni specificate dallo sviluppatore.

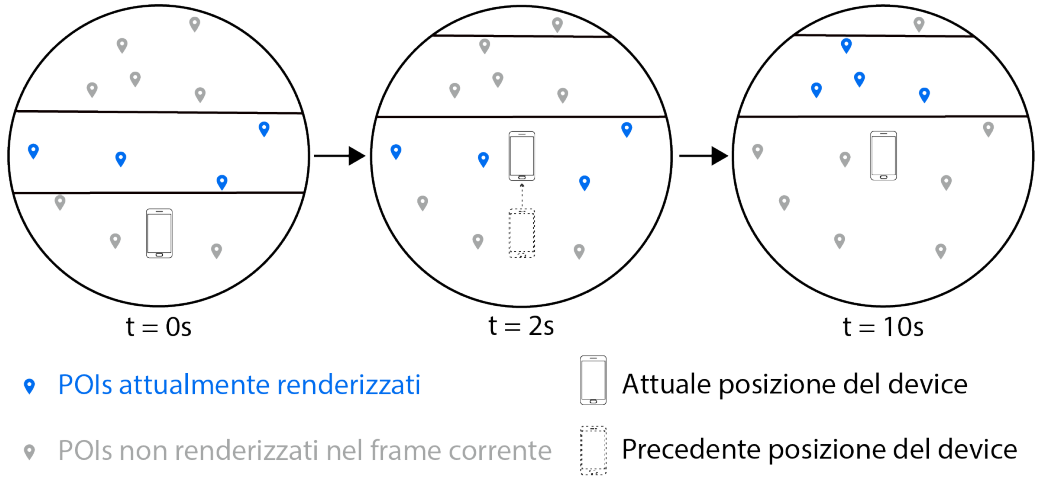

Figura 4.6: Problema di aggiornamento della scena quando almeno uno tra il *Near Plane Clipping* ed il *Far Plane Clipping* è abilitato. Nell'esempio il *Refresh Timer* aggiorna la scena ogni 10 secondi, quindi se l'utente si muove tra due aggiornamenti consecutivi non vedrà correttamente tutti i *POIs* che dovrebbe vedere.

Quindi, quando viene processato un *frame*, gli step che vengono eseguiti in successione per controllare se la scena è da aggiornare oppure no sono i seguenti:

1. se la scena è aggiornata, ovvero se il parametro booleano *sceneIsUpToDate* è settato a *true*, ed il *FOV Clipping* è abilitato, controlliamo se l'utente si sta guardando attorno utilizzando la differenza tra il *Bearing* attuale ed il *Bearing* del device al momento dell'ultimo aggiornamento della scena; in base al risultato di questo controllo modificheremo il valore di *sceneIsUpToDate* per indicare se la scena ha bisogno di essere aggiornata oppure no;

2. se la scena è aggiornata ed è abilitato almeno uno tra il *Near Plane Clipping* ed il *Far Plane Clipping*, allora controlliamo se l'utente si è spostato in modo significativo utilizzando la differenza tra la posizione GPS attuale e quella memorizzata durante l'ultimo aggiornamento della sce-

na; anche in questo caso, modificheremo il valore di *sceneIsUpToDate* di conseguenza;

3. a questo punto controlliamo se il valore di *sceneIsUpToDate* è *true* o *false*; nel primo caso la scena è aggiornata, quindi non effettuiamo più nessun calcolo fino al prossimo *frame*, mentre nel secondo caso andremo ad aggiornare la scena processando il *frame* corrente.

Una cosa importante da notare è che ogni n millisecondi il *Refresh Timer*, se abilitato dallo sviluppatore, andrà a modificare il valore di *isSceneUpToDate* forzando l'aggiornamento della scena.

Nel caso in cui la scena sia da aggiornare processando le informazioni ottenute al *frame* corrente, allora non resta che controllare quali sono i *POIs* visibili, quali sono i *Renderable* associati che devono essere renderizzati e dove devono essere posizionati nella scena. All'interno della libreria *AR POI Experiences* tutte queste operazioni sono finalizzate all'istanziazione del parametro di tipo *ArPoiAnchorNode* associato ad ogni *POI* visibile.

4.6 ArPoiAnchorNode

Come detto in precedenza, lo sviluppatore può creare una serie di *RealPoiNode* da aggiungere alla scena per poterli visualizzare in Realtà Aumentata, tutti memorizzati all'*ArPoiScene* in un apposito *ArrayList*[7]. La creazione dei contenuti virtuali associati ad ogni *POI* avviene nel momento in cui la scena si aggiorna, in modo tale che l'utente possa visualizzare sempre dei contenuti aggiornati in base ai propri spostamenti ed in base a come la scena è stata configurata. In particolare, ogni *RealPoiNode* ha un parametro di tipo *ArPoiAnchorNode* che, quando viene istanziato, si occupa della creazione e del corretto posizionamento del *Renderable* associato al *POI* all'interno del ciclo di rendering.

[7]Android APIs: ArrayList, `https://developer.android.com/reference/java/util/ArrayList`.

Tra le varie funzionalità fornite da *Sceneform* abbiamo la classe **AnchorNode**[8] che ci permette di posizionare automaticamente un contenuto virtuale nell'ambiente sfruttando una apposita *Anchor*. Per gli scopi della nostra tesi, come vedremo in seguito nel corso di questo capitolo, il posizionamento automatico fornito non era la soluzione ideale, motivo per cui abbiamo creato la classe *ArPoiAnchorNode* estendendo le funzionalità fornite da *AnchorNode*.

Per ogni *RealPoiNode* aggiunto alla scena dobbiamo innanzitutto controllare se deve essere visualizzato o meno in base a come la scena è configurata tramite i seguenti step:

- calcoliamo la distanza tra device e *POI*;
- se almeno uno tra il *Near Plane Clipping* ed il *Far Plane Clipping* è abilitato, allora dobbiamo controllare se il *POI* si trova all'interno dei limiti definiti dallo sviluppatore; in caso affermativo continuiamo con questi step, invece in caso negativo il *Renderable* non verrà creato;
- ora calcoliamo il *Bearing* del *POI*, ovvero l'angolo in gradi calcolato in senso orario, che c'è tra il Nord e la linea retta che collega la coordinata GPS del device a quella del *POI* preso in esame;
- conoscendo il *Bearing* del device e quello del *POI* possiamo poi calcolare l'angolo tra il device ed il *POI*, utile per posizionare correttamente il *Renderable* all'interno della scena;
- se nella scena è stato abilitato il *FOV Clipping*, utilizzando l'angolo appena calcolato possiamo controllare se il *POI* si trova all'interno del cono di vista definito oppure no; come nel caso precedente, in caso affermativo continuiamo seguendo questi step, altrimenti ci fermiamo;
- ora che abbiamo la distanza e l'angolo in gradi tra device e *POI*, possiamo ricavare le coordinate (x, y, z) in cui poter creare l'*Anchor* associata al *Renderable* che dobbiamo renderizzare;

[8]Sceneform APIs: AnchorNode, `https://developers.google.com/ar/reference/java/com/google/ar/sceneform/AnchorNode`.

- una volta creata l'*Anchor* non resta che istanziare l'*ArPoiAnchorNode* ed applicarci tutte le configurazioni specificate dallo sviluppatore.

Di seguito andremo a dare maggiori spiegazioni e dettagli su come questi passaggi sono stati implementati, motivando alcune delle scelte implementative più importanti che sono state fatte.

4.6.1 Distanza tra device e POI

La prima operazione necessaria è il calcolo della distanza presente tra il device ed il *POI* preso in considerazione. Per calcolarla esistono diverse tecniche ed algoritmi che si differenziano in base al grado di errore della distanza misurata ed in base alle performance.

Tra le tecniche analizzate ne sono state implementate due:

1. la *Formula dell'emisenoverso*[9], formula di trigonometria sferica che ci permette di calcolare la *distanza di cerchio massimo* fra due punti che si trovano su una sfera; nel nostro caso la utilizzeremo per calcolare la minore distanza in linea d'aria che collega due coordinate GPS;

2. la *Formula di Vincenty* [70], formula composta da due metodi iterativi che sfruttano un modello ellissoidale della Terra; il primo, chiamato *Metodo Diretto*, serve per calcolare la coordinata GPS di un particolare punto data una particolare angolazione ed una distanza da un'altra coordinata, mentre il secondo, chiamato *Metodo Inverso*, viene utilizzato per calcolare la lunghezza della *geodetica*[10] tra una coppia di coordinate GPS ed è proprio quello che abbiamo implementato.

È interessante notare la principale differenza tra questi due approcci: il primo utilizza un modello sferico della Terra, il che semplifica i calcoli necessari ma

[9]Formula dell'emisenoverso, `http://www.movable-type.co.uk/scripts/latlong.html`, `https://community.esri.com/groups/coordinate-reference-systems/blog/2017/10/11/vincenty-formula`.

[10]Con il termine *geodetica* si intende la curva di minor lunghezza che congiunge due punti di uno spazio e giace sullo spazio stesso.

allo stesso tempo introduce un errore maggiore, mentre il secondo approccio utilizza un modello ellissoidale della Terra, tenendo così in considerazione la deformazione dovuta alla rotazione terrestre, ed un algoritmo iterativo che riduce l'errore ad ogni iterazione in cambio di un costo computazionale maggiore.

Prendiamo ora in considerazione la *Formula dell'emisenoverso*: dati due punti generici sulla superficie terrestre, definiamo la latitudine di un generico punto X come ϕX, la longitudine di un generico punto X come λX ed il raggio medio della Terra R. L'algoritmo utilizzato dalla *Formula dell'emisenoverso* è definito semplicemente da 3 passaggi:

$$a = \sin^2\left(\frac{\phi B - \phi A}{2}\right) + \cos(\phi A) \cdot \cos(\phi B) \cdot \sin^2\left(\frac{\lambda B - \lambda A}{2}\right); \tag{4.1}$$

$$c = 2 \cdot \text{atan2}\left(\sqrt{a}, \sqrt{1-a}\right); \tag{4.2}$$

$$d = R \cdot c; \tag{4.3}$$

L'equazione 4.1 corrisponde al calcolo del quadrato di metà della lunghezza della geodetica che collega i due punti A e B, risultato utilizzato successivamente per calcolare il valore di c nell'equazione 4.2, il quale corrisponde alla distanza angolare in radianti tra i due punti, mentre d è la distanza in chilometri calcolata in linea d'aria tra A e B. In Figura 4.7 possiamo notare che R è rappresentato come raggio terrestre all'equatore, pari a circa 6378 km, mentre nel nostro caso abbiamo utilizzato il raggio terrestre medio che viene calcolato tramite una media delle distanze di tutti i punti del globo dal centro della Terra alla superficie ed è pari a circa 6371 km.

Per quanto riguarda invece il *Metodo Inverso* della *Formula di Vincenty* i calcoli sono leggermente più complessi e vengono iterati più volte al fine di far convergere il risultato fino ad un certo grado di accuratezza. Tutti i dettagli sulle formule utilizzate per questo metodo sono descritti in maniera approfondita in Appendice C.

Dopo averli implementati e testati entrambi abbiamo optato per l'utilizzo della *Formula dell'emisenoverso* per diversi fattori, in particolare:

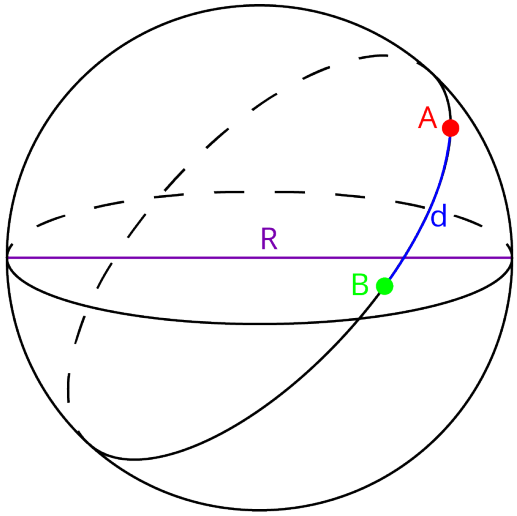

Figura 4.7: Distanza di cerchio massimo d tra due punti A e B sulla superficie terrestre, assumendo che la Terra abbia forma sferica.

1. il grado di errore per questo metodo è pari a circa lo 0.3%, il che significa che per calcolare la distanza tra due coordinate GPS molto distanti fra loro, ad esempio una distanza superiore ai 400 km, otterremo un errore che per molte applicazioni è significativo; nel nostro caso, ci sembra poco probabile che la libreria *AR POI Experiences* venga utilizzata per la creazione di esperienze in Realtà Aumentata utilizzando punti di interesse così distanti dall'utente, per cui l'errore ottenuto da questo metodo risulterebbe trascurabile;

2. il *Metodo Inverso* ha un costo computazionale molto più elevato rispetto alla *Formula dell'emisenoverso*; dato che stiamo sviluppando una possibile soluzione da utilizzare su mobile, abbiamo preferito mantenere delle prestazioni elevate sacrificando un piccola percentuale di accuratezza nella distanza ottenuta.

Come accennato in precedenza, questa distanza ci sarà successivamente utile per effettuare tutti i controlli relativi al *Clipping* e per la corretta creazione dell'*Anchor* a cui poter assegnare il contenuto virtuale da renderizzare all'interno della scena.

Distanza limite all'interno di ARCore

Durante lo sviluppo della libreria *AR POI Experiences* ci siamo imbattuti in un limite imposto dagli sviluppatori di *ARCore* che riguarda la distanza massima in metri a cui poter posizionare contenuti virtuali all'interno di una scena in Realtà Aumentata. Questo limite non è descritto nella documentazione ufficiale, ma a seguito di numerosi test ci siamo accorti che posizionando un contenuto virtuale ad una distanza superiore ai 20 metri dal device, il contenuto non veniva visualizzato all'interno della scena, molto probabilmente perché per *ARCore* risulta difficile tracciare la *Pose* di contenuti così distanti dalla fotocamera.

Per risolvere questa problematica abbiamo deciso di aggiungere un semplice controllo sulla distanza calcolata tra device e *POIs*, definendo così una *distanza di rendering* limitata ad un massimo di 20 metri dal device, mentre per tutti i calcoli descritti nei prossimi punti abbiamo utilizzato la distanza reale appena calcolata. Grazie alla *distanza di rendering* possiamo far sì che tutti i contenuti virtuali, anche i più distanti dall'utente, siano visibili all'interno della scena in Realtà Aumentata.

4.6.2 Angolo tra device e POI

Altro calcolo fondamentale per creare l'*Anchor* ed ottenere un corretto posizionamento del *Renderable* è proprio il calcolo dell'angolo tra il device ed il *POI* (Figura 4.8). Per poterlo calcolare abbiamo bisogno di due informazioni: il *Bearing*[11] del device e quello del *POI*.

Il *Bearing* del device è una informazione che possediamo già grazie all'utilizzo della libreria *Device Orientation* spiegata nel capitolo precedente, mentre il *Bearing* del *POI* preso in considerazione dobbiamo calcolarcelo. A tal scopo abbiamo implementato un apposito metodo all'interno della classe

[11]Ricordiamo che con *Bearing* intendiamo l'angolo in gradi che si forma tra una coordinata GPS ed il Nord magnetico ed è calcolato in senso orario.

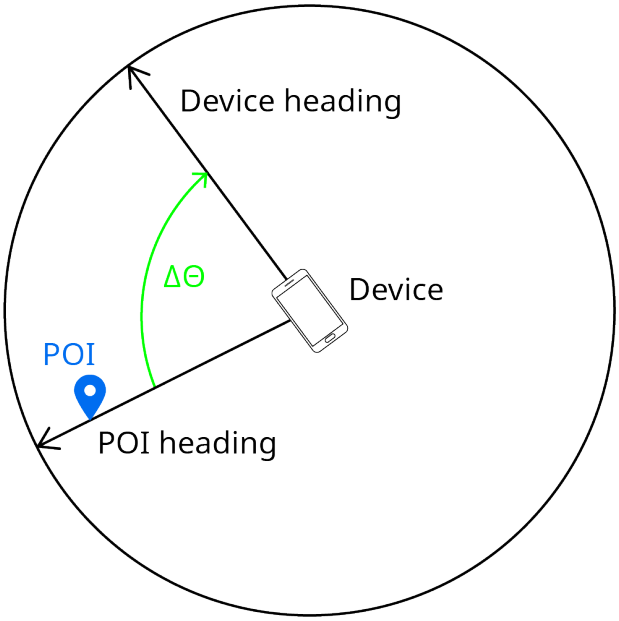

Figura 4.8: Angolo formato tra il device ed un *POI*, indicato con $\Delta\Theta$.

ArPoiMathUtils, la quale fornisce tutti i metodi necessari per le operazioni da eseguire, compreso il calcolo della distanza descritto precedentemente.

Il *Bearing* del *POI* si ottiene calcolando l'angolo che abbiamo tra il Nord ed una linea immaginaria, indicata in Figura 4.8 come *POI Heading*, che tracciamo seguendo la distanza di cerchio massimo tra la posizione GPS del device e quella del *POI* preso in considerazione (Figura 4.9).

Indichiamo con Θ_p il *Bearing* del *POI*, con ϕX la latitudine in radianti di un punto X e con λX la sua longitudine in radianti, con $\Delta\lambda$ indichiamo la differenza tra la longitudine di due punti mentre A e B sono, rispettivamente, la posizione GPS del device e quella del *POI* che stiamo considerando. La formula utilizzata per ottenere il Bearing del *POI* è la seguente:

$$a = \sin\Delta\lambda \cdot \cos\phi B; \tag{4.4}$$

$$b = \cos\phi A \cdot \sin\phi B - \sin\phi A \cdot \cos\phi B \cdot \cos\Delta\lambda; \tag{4.5}$$

$$\Theta_p = \text{atan2}\,(a, b)\,; \tag{4.6}$$

Dato che abbiamo bisogno di un angolo in gradi, dobbiamo ricordarci di

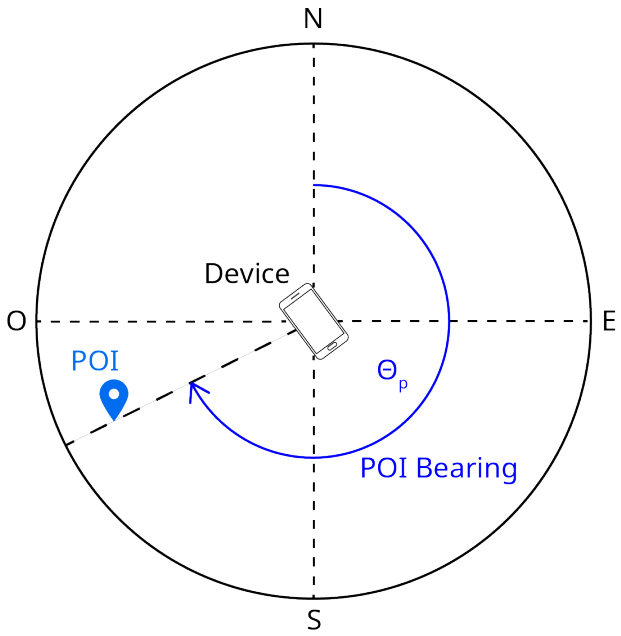

Figura 4.9: *Bearing* del *POI*, indicato con Θ_p.

convertire il risultato ottenuto da atan2, che sarà in radianti e compreso tra $[-\pi, +\pi]$, nuovamente in gradi e normalizzarlo nell'intervallo $[0.0, 360.0]$.

Una volta calcolato il *Bearing* del *POI* in gradi non ci resta che calcolare qual è la differenza tra i due *Bearing* per ottenere l'angolo tra il device ed il *POI* (Figura 4.10). Indichiamo questa differenza con $\Delta\Theta$, il *Bearing* del device con Θ_d mentre quello del *POI* con Θ_p:

$$\Delta\Theta = (\Theta_p + 360 - \Theta_d)\,\%360; \tag{4.7}$$

Come per il calcolo della distanza tra *POI* e device, anche il calcolo dell'angolo tra i due è principalmente finalizzato alla creazione dell'*Anchor* ed al corretto posizionamento del *Renderable* associato al *POI*, ma è anche utilizzato per controllare se il contenuto virtuale deve essere renderizzato o meno nel caso in cui lo sviluppatore abbia abilitato il *FOV Clipping*.

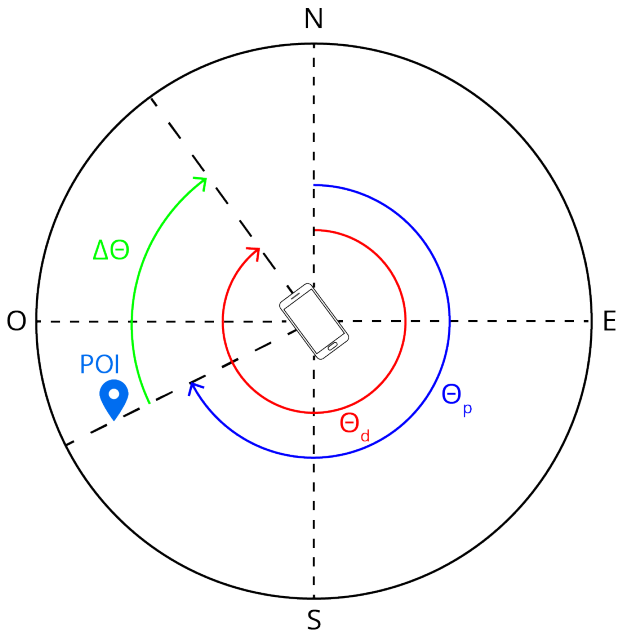

Figura 4.10: Calcolo dell'angolo $\Delta\Theta$ tra *POI* e device (colore verde). Θ_p è il *Bearing* del *POI* (colore blu), mentre Θ_d è il *Bearing* del device (colore rosso).

4.6.3 Creazione dell'Anchor

Dopo aver calcolato la distanza e l'angolo tra il device ed il *POI* che stiamo considerando, supponendo che abbia superato i controlli sul *Clipping* e debba essere renderizzato, non resta che creare l'*Anchor* a cui poter finalmente associare il *Renderable*.

L'*Anchor* viene creata in un particolare punto del mondo in Realtà Aumentata utilizzando specifiche coordinate (x_a, y_a, z_a) che otteniamo tramite semplici operazioni di rotazione sugli assi:

$$x_a = d \sin(\Delta\Theta); \tag{4.8}$$

$$y_a = y_{camera} + \Delta y; \tag{4.9}$$

$$z_a = -d \cos(\Delta\Theta); \tag{4.10}$$

Ricordiamo che con d indichiamo la distanza tra device e *POI* e con $\Delta\Theta$ l'angolo tra i due punti. Per calcolare y utilizziamo la fotocamera come ri-

ferimento, la cui posizione è una delle informazioni contenute all'interno del *frame* e la possiamo ottenere grazie alla primitiva *getDisplayOrientedPose()*[12] fornita da *ARCore*. Una volta ottenuta la posizione, sommiamo alla coordinata y della fotocamera (y_{camera}), un parametro definito dallo sviluppatore (Δy): di default le *Anchor* e di conseguenza i *Renderable* saranno creati alla stessa altezza della fotocamera, ma lo sviluppatore può decidere arbitrariamente se posizionarli più in basso oppure più in alto.

Le rotazioni che abbiamo utilizzato sono ispirate alla rotazione di una matrice bidimensionale:

$$\begin{bmatrix} x_a \\ z_a \end{bmatrix} = \begin{bmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{bmatrix} \begin{bmatrix} x \\ z \end{bmatrix}; \tag{4.11}$$

$$x_a = x\cos\alpha - z\sin\alpha; \tag{4.12}$$

$$z_a = x\sin\alpha + z\cos\alpha; \tag{4.13}$$

ed è interessante notare che nel nostro particolare caso è stato possibile semplificare ulteriormente i calcoli poiché non abbiamo bisogno della componente x nella trasformazione:

$$x = 0; \qquad z = -d; \qquad \alpha = \Delta\Theta; \tag{4.14}$$

$$\begin{aligned} x_a &= 0\cos(\Delta\Theta) - (-d)\sin(\Delta\Theta) \\ &= d\sin(\Delta\Theta); \end{aligned} \tag{4.15}$$

$$\begin{aligned} z_a &= 0\sin(\Delta\Theta) + (-d)\cos(\Delta\Theta) \\ &= -d\cos(\Delta\Theta); \end{aligned} \tag{4.16}$$

Nell'equazione 4.14 possiamo notare che la distanza tra device e *POI* viene utilizzata con segno invertito, fondamentale per posizionare correttamente l'*Anchor* tenendo in considerazione l'orientamento degli assi del sistema di coordinate del mondo in AR.

[12] ARCore APIs: Camera getDisplayOrientedPose, `https://developers.google.com/ar/reference/java/com/google/ar/core/Camera#getDisplayOrientedPose()`.

4.6.4 Creazione dell'ArPoiAnchorNode

Finalmente ora abbiamo tutte le informazioni necessarie per poter istanziare il parametro di tipo *ArPoiAnchorNode* del *POI* che stiamo considerando. I passaggi di cui abbiamo bisogno variano leggermente in base alle configurazioni specificate dallo sviluppatore, in particolare tra le funzionalità offerte dalla libreria abbiamo due diverse modalità per aggiornare graficamente la posizione di un *Renderable* all'interno della scena:

1. in **Modalità Teleport** quando la scena si aggiorna tutti i *Renderable* vengono rimossi dalla scena per poi ricrearli nuovamente nella posizione aggiornata;

2. invece di essere rimossi, in **Modalità Linear** definiamo una transizione lineare che sposti il *Renderable* dalla vecchia posizione a quella nuova.

Se lo sviluppatore non specifica quale delle due modalità utilizzare, di default la libreria *AR POI Experiences* utilizzerà la *Modalità Teleport*.

In entrambi i casi, per istanziare il parametro di tipo *ArPoiAnchorNode* abbiamo bisogno di alcune informazioni, ovvero un *Renderable* da renderizzare, una *Anchor* a cui associare il *Renderable*, un *Trackable* a cui associare l'*Anchor* e le configurazioni definite dallo sviluppatore nel *RealPoiNode*. Tutte queste informazioni, arrivati a questo punto, le possediamo già: il *Renderable* e le configurazioni sono tutti parametri settati dallo sviluppatore in fase di creazione di un *RealPoiNode*, l'*Anchor* è stata creata allo step precedente sfruttando la distanza e l'angolo tra *POI* e device, mentre il *Trackable* in questo caso sarà l'intera scena in AR.

In Figura 4.11 possiamo vedere la struttura generale dello *Scene Graph*[13] nel nostro caso, dove il nodo radice è rappresentato dalla scena in Realtà Aumentata, tutti gli *ArPoiAnchorNode* sono nodi figli della scena ed ognuno di essi avrà il suo *RenderableNode* sfruttato per il rendering del *Renderable*,

[13]Ricordiamo che lo *Scene Graph* è la struttura ad albero che *Sceneform* utilizza per la creazione di tutti i contenuti virtuali nell'ambiente in Realtà Aumentata.

il quale a sua volta potrebbe avere o non avere un nodo figlio che rappresenta il *ViewRenderable* associato.

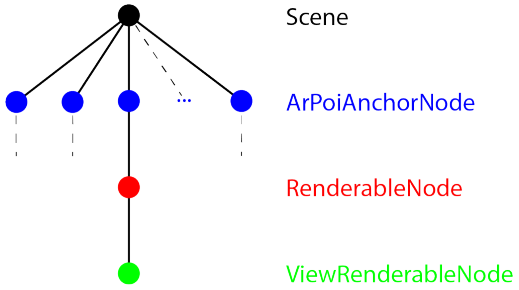

Figura 4.11: Struttura dello *Scene Graph* creato utilizzando la libreria *AR POI Experiences*.

Aggiornamento con Modalità Teleport

Nel caso in cui la scena sia stata configurata utilizzando la *Modalità Teleport*, quello che dobbiamo fare è applicare la procedura di **detach** al *RealPoiNode*, ovvero dobbiamo disabilitare il rendering del contenuto virtuale e rimuovere l'*Anchor* dalla scena in modo tale che *ARCore* non si preoccupi più di aggiornare la relativa *Pose*. Come indicato all'interno della documentazione ufficiale di *ARCore*, la procedura di *detach* di un nodo nella scena è fondamentale per evitare una riduzione drastica delle prestazioni, questo perchè, come detto anche in precedenza, *ARCore* modifica ed aggiorna la propria conoscenza dell'ambiente reale con il passare del tempo e con l'acquisizione di nuovi *feature points*, informazioni che utilizzerà per aggiornare anche la *Pose* di ogni *Anchor* creata all'interno di una scena. Inoltre, per scelta implementativa degli sviluppatori di *ARCore*, non è possibile muovere o spostare un'*Anchor*, motivo per cui l'unico modo che abbiamo per aggiornare la posizione di un *Renderable* associato ad un *POI* è quello di creare una nuova *Anchor* e rimuovere dalla scena quella precedente ormai inutilizzata.

Una volta effettuato il *detach* non resta che istanziare e configurare correttamente l'*ArPoiAnchorNode* in base alle preferenze specificate dallo sviluppatore in fase di creazione del relativo *RealPoiNode*:

- abilitare o disabilitare la riduzione della dimensione del *Renderable* in base alla distanza tra device e *POI*; se abilitato, i contenuti virtuali dei *POI* più vicini saranno leggermente più grandi rispetto a quelli in lontananza;

- se aggiunto, creare e visualizzare il *ViewRenderable*;

- se abilitato, eseguire il comportamento in risposta al *tap* dell'utente sul *Renderable*;

- se abilitato, eseguire il comportamento specificato in fase di rendering.

Aggiornamento con Modalità Linear

Invece nel caso in cui la scena sia stata configurata utilizzando la *Modalità Linear*, non applichiamo la procedura di *detach* immediatamente ma aggiorniamo le informazioni e le configurazioni associate alla precedente istanza dell'*ArPoiAnchorNode*. Questo perché per realizzare la transizione lineare dal punto precedente alla nuova posizione abbiamo bisogno di entrambe le *Anchor*.

La transizione lineare viene creata utilizzando un **ObjectAnimator**[14], classe fornita dalle *Android APIs* per aggiungere animazioni a specifici oggetti che ci garantisce il vantaggio di poter realizzare ed applicare qualsiasi tipo di animazione ai nostri *Renderable*. Per gli scopi di questa tesi, l'unica animazione realizzata è appunto la transizione tra due *Anchor*, ma, al fine di facilitare un possibile sviluppo futuro di nuove features della libreria *AR POI Experiences*, abbiamo realizzato un'apposita classe, chiamata *AnimationUtils*, che contiene il metodo utilizzato per la transizione e che potrà contenere tanti metodi quante le animazioni che si implementeranno in futuro.

I passi implementati al fine di creare l'animazione ed applicarla al *Renderable* che stiamo aggiornando sono i seguenti:

[14]Android APIs: ObjectAnimator, `https://developer.android.com/reference/android/animation/ObjectAnimator`.

1. per prima cosa istanziamo un nuovo oggetto di tipo *ArPoiAnchorNode* e gli associamo la nuova *Anchor* calcolata in questo *frame*, ma non gli associamo nessun *Renderable* da visualizzare;

2. definiamo la durata dell'animazione in base a come la scena è stata configurata, in modo tale che il tempo utilizzato per effettuare la transizione sia minore o uguale al tempo impiegato dalla scena per aggiornarsi;

3. utilizziamo il metodo all'interno della classe *AnimationUtils* per creare una transizione dall'*Anchor* precedente a quella nuova;

4. avviamo la transizione.

Il passo 1, dato che in *ARCore* le transizioni non sono previste tra i comportamenti nativi di un *Renderable*, ci permette non solo di specificare qual è la destinazione della nostra transizione, ma anche di far sì che alla fine dell'animazione il *Renderable* rimanga associato all'*Anchor* creata in questo *frame*, in modo tale da poter effettuare il *detach* sull'*Anchor* precedente.

In Figura 4.12[15] possiamo vedere graficamente i passaggi appena descritti. In ordine da 1 a 6 abbiamo: l'*ArPoiAnchorNode* associato all'$Anchor_i$ calcolata durante l'ultimo aggiornamento della scena al $frame_i$, la creazione del nuovo *ArPoiAnchorNode* utilizzato come destinazione associato all'$Anchor_{i+n}$ appena creata nel $frame_{i+n}$, la creazione della transizione da $Anchor_i$ ad $Anchor_{i+n}$, l'avvio della transizione, l'arrivo a destinazione ed infine il processo di *detach* sull'$Anchor_i$ non più utilizzata.

Arrivati a questo punto, l'aggiornamento della nostra scena è terminato e tutti i *POIs* che hanno superato eventuali controlli sul *Clipping* avranno un *Renderable* posizionato correttamente e renderizzato all'interno nella scena, mentre tutti gli altri *POIs* che per qualche ragione non vengono renderizzati saranno sottoposti alla procedura di *detach* al fine di rimuovere *Anchor* non

[15]Icona di *Baby Groot* scaricata dalla piattaforma *Icons8*, `https://icons8.it/icon/set/groot/all`.

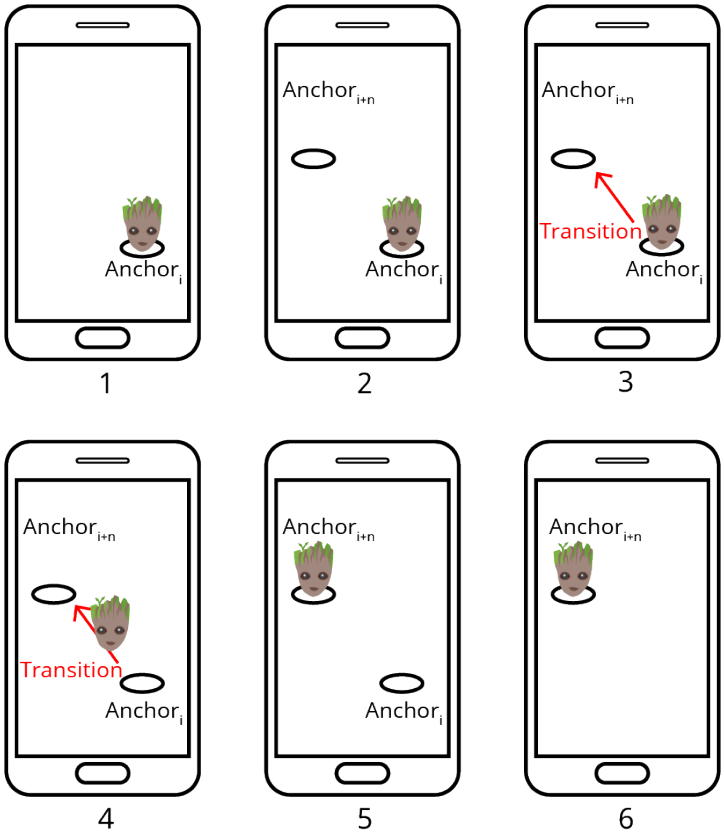

Figura 4.12: Esempio di aggiornamento della posizione di un *POI* utilizzando la *Modalità Linear*. Con $Anchor_i$ e $Anchor_{i+n}$ indichiamo, rispettivamente, l'*Anchor* creata al *frame* precedente e quella creata nel *frame* corrente ed in rosso abbiamo l'animazione per simulare la transizione.

utilizzate in questo *frame* che potrebbero appesantire *ARCore*, peggiorando l'esperienza dell'utente.

4.6.5 Ciclo di rendering

All'inizio di questo sottocapitolo abbiamo accennato al fatto che il posizionamento automatico fornito dalla classe *AnchorNode* di *Sceneform* non era la soluzione ideale per il nostro caso, motivo per cui abbiamo esteso la classe creando ciò di cui avevamo bisogno. Tra le funzionalità fornite nativamente

dalla classe *AnchorNode* abbiamo il metodo chiamato *onUpdate()*[16], all'interno del quale vengono effettuati tutti i calcoli necessari al fine di posizionare il *Renderable* in corrispondenza della posizione dell'*Anchor* associata.

Quando nelle prime versioni della libreria *AR POI Experiences* utilizzavamo direttamente la classe *AnchorNode* per gestire tutta la parte di rendering dei contenuti ci siamo accorti che tutti i *Renderable* era posizionati correttamente, ma erano orientati male. Come possiamo vedere in Figura 4.13, questo causava una riduzione della *User Experience*, cosa che volevamo assolutamente evitare.

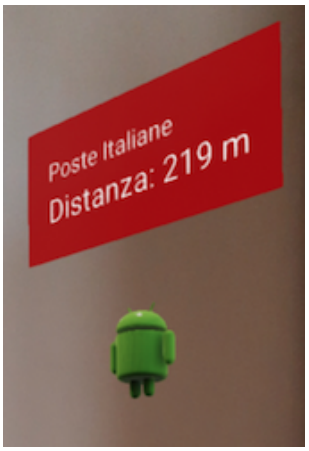

Figura 4.13: Problema causato dal posizionamento automatico di un *Renderable* utilizzando la classe *AnchorNode*.

La classe *ArPoiAnchorNode* descritta finora è nata proprio per porre rimedio a questo problema: grazie all'*override*[17] del metodo *onUpdate()* abbiamo implementato tutte le traslazioni e rotazioni necessarie ad un corretto posizionamento del *Renderable* nei confronti della posizione dell'utente. Per

[16]Sceneform APIs: AnchorNode onUpdate, `https://developers.google.com/ar/reference/java/com/google/ar/sceneform/AnchorNode#onUpdate(com.google.ar.sceneform.FrameTime)`.

[17]Il termine *override* è utilizzato nella programmazione ad oggetti per indicare l'operazione di modifica o estensione delle funzionalità di un metodo ereditato dalla superclasse.

ottenere il miglioramento desiderato abbiamo implementato i passi descritti di seguito:

1. ricaviamo le coordinate cartesiane della fotocamera e dell'*Anchor* all'interno della scena in Realtà Aumentata;
2. otteniamo il vettore direzione tramite una semplice sottrazione tra le due coordinate cartesiane ottenute;
3. utilizzando il vettore direzione e l'*up vector*, ovvero il vettore che indica dove si trova "l'alto" della camera, possiamo ottenere l'angolo da applicare al *Renderable* per far sì che sia rivolto verso l'utente;
4. scaliamo le dimensioni del *Renderable* in base alle configurazioni impostate dallo sviluppatore;
5. se è stato creato e configurato, assegniamo al *ViewRenderable* la stessa posizione ed orientazione del *Renderable*, in modo tale che siano coerenti;
6. infine, se è stato configurato, eseguiamo il *renderEvent* associato a questo *POI*.

In Figura 4.14 possiamo vedere i risultati ottenuti, grazie ai quali abbiamo sicuramente migliorato l'esperienza dell'utente nell'utilizzo di una applicazione sviluppata tramite questa libreria.

4.7 Conteggio dei POIs nella scena

Sfruttando i calcoli descritti precedentemente per la creazione di un *ArPoiAnchorNode*, abbiamo implementato anche una funzionalità per sapere quanti sono i *POIs* che si trovano attualmente nella scena e dove si trovano rispetto al device. In particolare, come possiamo vedere in Figura 4.15, abbiamo suddiviso l'area attorno al device in 3 differenti zone in cui un *POI* potrebbe trovarsi: a sinistra, di fronte oppure a destra del device. Questa

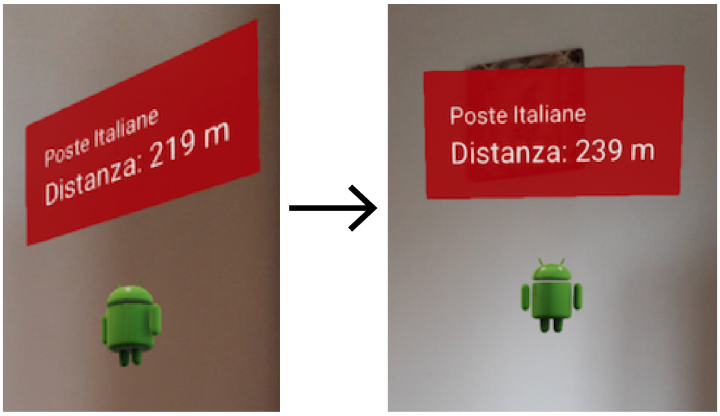

Figura 4.14: Miglioramento ottenuto rispetto al posizionamento automatico dei *Renderable*.

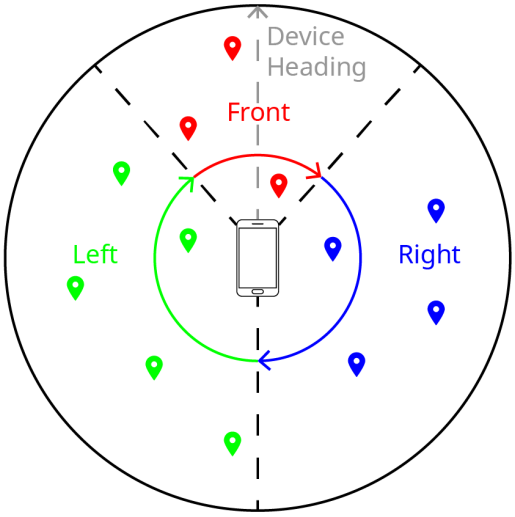

Figura 4.15: Zone in cui si può trovare un *POI* rispetto alla posizione ed all'orientazione del device. In rosso i *POIs* considerati di fronte al device, in verde quelli considerati alla sua sinistra ed in blu quelli considerati alla sua destra.

funzionalità risulta utile nel caso in cui lo sviluppatore voglia fornire agli utenti della propria applicazione delle maggiori indicazioni su come trovare i *POIs* nella scena. Ad esempio, se prendiamo il caso in cui l'app abbia lo scopo di mostrare informazioni sui migliori ristoranti di una città tramite la

Realtà Aumentata, allora all'interno della *UI*[18] in cui è visualizzata la scena in AR potrebbe esserci un'indicazione che mostra all'utente quanti sono i ristoranti alla sua sinistra e quanti alla sua destra.

Quando abbiamo realizzato questa funzionalità, ci siamo anche accorti che all'interno di una scena potrebbero esserci *POIs* renderizzati, ovvero *POIs* il cui *Renderable* è attualmente visibile nella scena, ma anche *POIs* non renderizzati, ad esempio perché rimossi dalla scena per via del *Clipping* o del *Clustering*. Per dare un numero maggiore di informazioni allo sviluppatore ed agli utenti, abbiamo deciso di implementare la possibilità di ottenere qual è il numero di *POIs* visibili e di quelli non visibili in una delle tre zone.

4.8 Tracking State

Una cosa importante da ricordare quando si sviluppa un'applicazione in Realtà Aumentata è che, affinché tutto funzioni correttamente, devono esserci dei presupposti che vanno controllati e gestiti direttamente dallo sviluppatore, ad esempio all'interno della singola *Activity* prima di poter processare il *frame* corrente sarebbe meglio assicurarsi che il *frame* esista e non sia nullo, altrimenti si rischia di incorrere in errori e causare un crash dell'applicazione.

Questi presupposti potrebbero variare in base a quali tecnologie si stanno utilizzando. Nel nostro particolare caso, utilizzando *ARCore* e *Sceneform*, è necessario controllare qual è il cosiddetto **Tracking State**[19] della fotocamera, ovvero dobbiamo controllare se nel *frame* corrente la fotocamera del device è abilitata e sta ricevendo correttamente le immagini dall'ambiente reale. Esistono 3 diversi *Tracking State*:

1. *PAUSED*: *ARCore* ha smesso di tracciare l'ambiente reale, ma nel futuro potrebbe attivarsi ed iniziare di nuovo a tracciarlo;

[18] L'acronimo *UI* sta per *User Interface*.

[19] ARCore APIs: TrackingState, `https://developers.google.com/ar/reference/java/com/google/ar/core/TrackingState`.

2. *STOPPED*: *ARCore* ha smesso definitivamente di tracciare l'ambiente reale;

3. *TRACKING*: *ARCore* sta tracciando correttamente l'ambiente reale in questo istante.

Se, ad esempio, mentre stiamo utilizzando una generica applicazione realizzata con *ARCore* decidiamo di oscurare la fotocamera con un dito oppure inquadriamo una superficie completamente bianca, quello che succederà è che *ARCore* non riuscendo più a riconoscere nessun *feature points* passerà dallo stato di *TRACKING* a quello di *PAUSED*, informando così lo sviluppatore che al momento non è in grado di capire nulla dell'ambiente che viene inquadrato. Sulla base di queste informazioni dovrà essere proprio lo sviluppatore a gestire i 3 particolari casi in base alle proprie preferenze ed alle proprie necessità. A tal scopo, una delle scelte implementative è stata proprio quella di non gestire esplicitamente nessuno di questi casi all'interno della libreria *AR POI Experiences* per poter lasciare il libero arbitrio allo sviluppatore, evitando così di limitarne le possibili azioni.

4.8.1 Cleanup della scena

Durante alcuni test svolti nel corso dello sviluppo della libreria *AR POI Experiences* ci siamo accorti che quando *ARCore* non si trova nello stato di *TRACKING*, non riconoscendo un numero sufficiente di *feature points*, tutte le *Pose* e le *Anchor* perdono le loro proprietà di immutabilità nello spazio. Questo ovviamente capita proprio perchè *ARCore*, senza le informazioni sull'ambiente reale e sulla sua composizione, non può continuare a mantenere i contenuti virtuali nelle loro posizioni originarie.

Nelle prime versioni della soluzione sviluppata, il passaggio dallo stato di *TRACKING* ad uno degli altri due stati causava quindi un bug visivo per cui tutti i *Renderable* che si trovavano sullo schermo in quel momento non venivano rimossi dalla scena, ma continuavano a restare statici sullo schermo indipendentemente dai movimenti dell'utente. Per evitare questo comporta-

mento e migliorare l'esperienza creata per l'utente, tra le funzionalità fornite all'interno della libreria *AR POI Experiences* abbiamo implementato anche una procedura di **clean up** della scena che consiste semplicemente nel *detach* di tutte le *Anchor* presenti e nel disabilitare il rendering di tutti i contenuti virtuali. Tutto quello che lo sviluppatore dovrà fare all'interno delle proprie *Activity* sarà aggiungere questo semplice controllo ad ogni *frame*:

```
1 /* Il parametro frame rappresenta il frame corrente, mentre
2  * arPoiScene e' la scena in Realta' Aumentata.
3  */
4 if (frame.getCamera().getTrackingState() != TRACKING)
5   arPoiScene.cleanUp();
```

4.9 Clustering

Nel corso dello sviluppo della libreria e dei test eseguiti su prototipi ci siamo accorti che uno dei problemi principali che riduceva la *User Experience* era la sovrapposizione di due o più contenuti virtuali. Come possiamo vedere in Figura 4.16[20], questo scenario può capitare in diversi casi, per esempio due o più *POIs* potrebbero avere una coordinata GPS molto vicina oppure potrebbero essere molto distanti ma esattamente sulla stessa linea d'aria. Per ovviare a questo problema abbiamo optato per implementare il cosiddetto *Clustering*, che può essere definito come la ricerca di gruppi di oggetti, detti *Cluster*, da poter raggruppare in base, ad esempio, a fattori di similarità o di correlazione tra questi oggetti. Nel nostro specifico caso, gli oggetti saranno i *Renderable* di ogni *POI* e quando all'interno della nostra scena ce ne saranno due o più sovrapposti andremo a creare un *Cluster* che li contenga.

Come tutte le funzionalità fornite da *AR POI Experiences*, anche il *Clustering* è attivabile o disattivabile in base alle preferenze dello sviluppatore. Nel caso in cui venga attivato, l'operazione di *Clustering* viene fatta dopo la

[20]Icona del logo di Android scaricata dalla piattaforma *Icons8*, `https://icons8.com/icon/set/android/all`. Immagine usata come sfondo nel device scaricata dal sito `http://www.availableideas.com/30-amazing-illustrations-iphone-wallpapers/`.

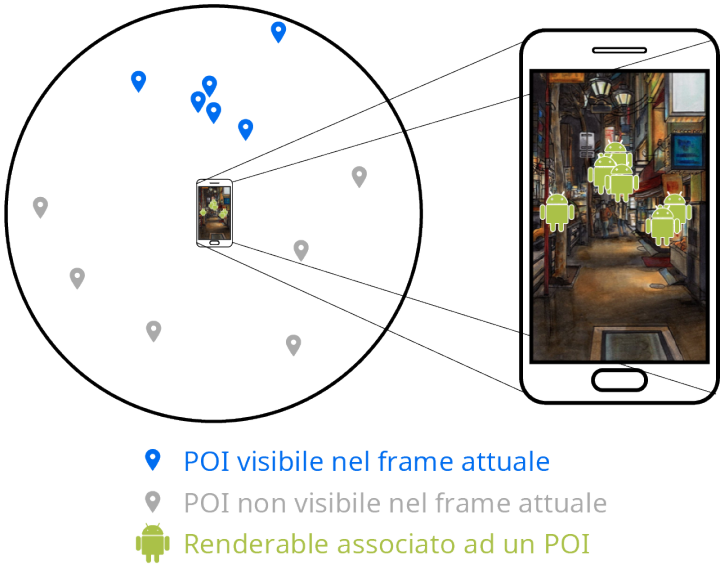

Figura 4.16: Sovrapposizione di contenuti virtuali all'interno della scena. Il gruppo centrale è il caso di *POIs* molto vicini geograficamente, mentre il gruppo a destra è il caso di *POIs* distanti geograficamente ma sulla stessa linea di visuale dell'utente.

creazione dei *Renderable* nel corso dell'aggiornamento della scena. Quindi, per tutti i *POIs*, andiamo ad effettuare i seguenti step al fine di individuare quali sono i contenuti sovrapposti:

1. presa la lista dei *RealPoiNode*, ovvero di tutti i *POIs* contenuti all'interno della nostra scena, controlliamo quali sono quelli che hanno superato eventuali controlli sul *Clipping* ed hanno un *Renderable* configurato e renderizzato correttamente nella scena;

2. successivamente controlliamo quali sono i *Renderable* sovrapposti nella scena al *frame* corrente e creiamo un *Cluster* per ogni gruppo di *Renderable* sovrapposti;

3. per ogni *POIs* che si trova all'interno di un *Cluster* disabilitiamo il rendering del relativo contenuto virtuale, in modo tale che l'unico *Ren-*

derable che sarà presente nella scena sarà quello relativo al *Cluster* creato;

4. infine procediamo alla creazione del *Renderable* da associare ad ogni *Cluster* creato.

In Figura 4.17[21] possiamo vedere una riproduzione grafica dei 4 step appena descritti.

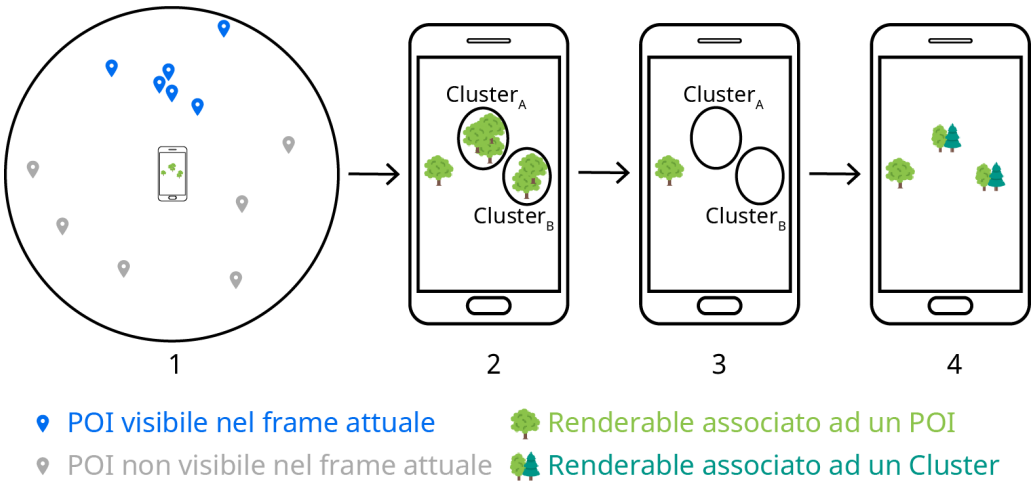

Figura 4.17: Passi realizzati per applicare il *Clustering*.

Nello step 1, per capire quali sono i *Renderable* attualmente visibili nella scena al *frame* corrente, ci basta un semplice controllo su ogni *POI*: tutti quelli renderizzati avranno un *ArPoiAnchorNode* istanziato e configurato, mentre gli altri no. Lo step 2 invece viene fatto utilizzando una delle APIs messe a disposizione da *ARCore*, ovvero il metodo *overlapTestAll(Node)*[22] legato alla scena in Realtà Aumentata. Questo metodo ci permette di testare

[21] Icone degli alberi scaricate dalla piattaforma *Icons8*, `https://icons8.com/icon/18047/simple-tree`, `https://icons8.com/icon/17593/pine-tree`.

[22] ARCore APIs: Scene overlapTestAll, `https://developers.google.com/ar/reference/java/com/google/ar/sceneform/Scene#overlapTestAll(com.google.ar.sceneform.Node)`.

se i limiti[23] di un generico *Node* nella nostra scena sono sovrapposti a quelli di uno o più altri *Node* e ci restituisce una lista con tutti i *Node* sovrapposti. Nel nostro caso i *Node* sono tutti i contenuti virtuali che si trovano all'interno della scena, ovvero sia i *Renderable* che i *ViewRenderable* che lo sviluppatore può associare o meno ad ogni *POI*. Dato che i *ViewRenderable* sono intesi come contenuti virtuali 2D da aggiungere al *Renderable* principale di ogni *POI* per poter fornire informazioni o azioni aggiuntive, dobbiamo filtrare la lista ottenuta come risultato dal metodo *overlapTestAll(Node)* al fine di controllare se il *Renderable* preso in esame è sovrapposto oppure no ad altri *Renderable*. Anche in questo caso, le APIs di *ARCore* ci vengono in aiuto fornendoci il metodo booleano *isTopLevel()*[24] che restituisce *true* nel caso in cui il *Node* preso in esame è considerato *Top Level*, altrimenti *false*. Un *Node* viene considerato come *Top Level* solo in due casi:

1. non ha nessun padre all'interno dello *Scene Graph*, ovvero è il nodo radice;

2. il padre del *Node* è la scena in Realtà Aumentata, ovvero il nodo padre è esattamente il nodo radice dello *Scene Graph*.

Come abbiamo visto in Figura 4.11, nel nostro caso tutti i *ViewRenderable* sono figli di un *RenderableNode*, tutti i *Renderable* sono figli di un *ArPoiAnchorNode* e tutti gli *ArPoiAnchorNode* sono figli della scena, ovvero, ad esclusione della scena in Realtà Aumentata, sono gli unici *Node* considerati *Top Level* all'interno del nostro *Scene Graph*. Utilizzando tutte queste informazioni, per poter rimuovere dalla lista di risultati restituita dal metodo *overlapTestAll(Node)* tutti i *ViewRenderable*, non ci resta che prendere ogni *Node* dalla lista e controllare se il suo nodo padre è considerato *Top Level* oppure no:

[23] Quando parliamo di *limiti di un Node* ci riferiamo al *Bounding Box* dell'oggetto, ovvero al parallelepipedo che contiene tutti i punti del contenuto virtuale.

[24] ARCore APIs: Node isTopLevel, `https://developers.google.com/ar/reference/java/com/google/ar/sceneform/Node#isTopLevel()`.

- tutti i *Renderable* hanno come padre un *ArPoiAnchorNode*, che è considerato come *Top Level*;

- invece tutti i *ViewRenderable* hanno come padre un *RenderableNode*, che non è considerato come *Top Level* e di conseguenza non sarà utilizzato per i controlli sulla sovrapposizione dei contenuti.

A questo punto, completati gli step 2, 3 e 4, ci ritroviamo nella situazione in cui abbiamo un *Cluster* per ogni gruppo di *POIs* il cui *Renderable* era sovrapposto ad altri ed abbiamo disabilitato il rendering di tutti questi *Renderable*, quindi quello che resta da fare è creare il contenuto virtuale associato ad ogni *Cluster* per poterlo mostrare all'interno della scena.

4.9.1 Creazione del Renderable di un Cluster

Per farlo utilizziamo la stessa procedura che applichiamo ad ogni *POI* descritta precedentemente: calcoliamo la distanza e l'angolo tra device e *Cluster*, controlliamo se supera eventuali test sul *Clipping*, creiamo l'*Anchor* e istanziamo l'*ArPoiAnchorNode*. Sorgono però alcune problematiche poiché per eseguire questi step abbiamo bisogno di una informazione fondamentale che al momento non conosciamo, ovvero la posizione GPS di ogni *Cluster*: senza questa informazione non possiamo calcolare ne la distanza ne l'angolo tra device e *Cluster* e di conseguenza non possiamo eseguire nessuno dei passi necessari alla creazione del relativo *Renderable*.

Per porre rimedio a questa problematica abbiamo implementato all'interno della libreria due diversi metodi[25] che determinano la posizione GPS da assegnare al *Cluster* sulla base dei *POIs* contenuti al suo interno:

1. *Geographic Midpoint*;

2. *Average Latitude-Longitude*.

[25]Calculation Methods: Geographic Midpoint and Average Latitude-Longitude, `http://www.geomidpoint.com/calculation.html`.

Di seguito utilizzeremo la notazione ϕ_i e λ_i per indicare, rispettivamente, la latitudine e la longitudine in radianti di un generico punto i, le lettere x_i, y_i e z_i per indicare le coordinate cartesiane di un generico punto i e con il pedice $_{mid}$ ci riferiremo a tutto ciò che riguarda il punto medio geografico che vogliamo ottenere.

Il primo metodo consiste nel calcolare il centro di gravità per ogni *Location* dei *POIs* contenuti all'interno di un *Cluster*. Per prima cosa dobbiamo convertire la latitudine e la longitudine di ogni coordinata GPS in coordinate (x, y, z) sul piano cartesiano tangente alla superficie terrestre:

$$x_i = \cos(\phi_i) \cdot \cos(\lambda_i); \tag{4.17}$$

$$y_i = \cos(\phi_i) \cdot \sin(\lambda_i); \tag{4.18}$$

$$z_i = \sin(\phi_i); \tag{4.19}$$

Successivamente calcoliamo la coordinata media di queste coordinate cartesiane:

$$x_{mid} = \frac{\sum_{i=1}^{n} x_i}{n}; \tag{4.20}$$

$$y_{mid} = \frac{\sum_{i=1}^{n} y_i}{n}; \tag{4.21}$$

$$z_{mid} = \frac{\sum_{i=1}^{n} z_i}{n}; \tag{4.22}$$

Infine tracciamo una linea immaginaria h dal centro della Terra a questa nuova coordinata media appena calcolata per ricavare il punto in cui la linea si interseca con la superficie terrestre, il quale sarà proprio il *punto medio geografico* che volevamo ottenere:

$$\lambda_{mid} = \text{atan2}(y_{mid}, x_{mid}); \tag{4.23}$$

$$h = \sqrt{x_{mid}^2 + y_{mid}^2}; \tag{4.24}$$

$$\phi_{mid} = \text{atan2}(z_{mid}, h); \tag{4.25}$$

In Figura 4.18 possiamo vedere una semplice rappresentazione grafica di questi passaggi.

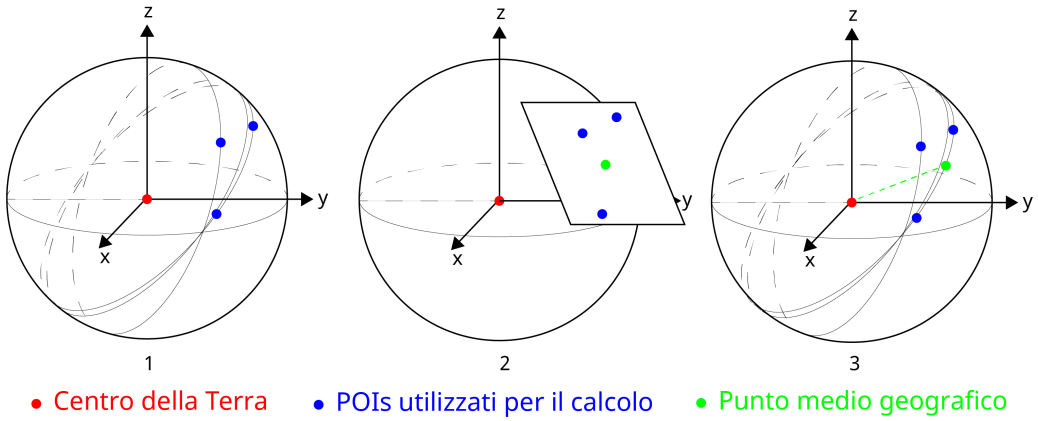

Figura 4.18: Calcolo della posizione GPS di *Cluster* tramite il metodo *Geographic Midpoint*. Nel passaggio 1 vediamo (in blu) i *POIs* che stiamo considerando per il calcolo, nel passaggio 2 la conversione da latitudine e longitudine a coordinate cartesiane sul piano tangente la superficie della Terra ed il punto medio calcolato (in verde) ed infine nel passaggio 3 otteniamo la coordinata GPS che stavamo cercando.

Il secondo metodo invece è una semplice media delle coordinate GPS di ogni *POI* in cui consideriamo la superficie terrestre come se fosse piatta e rettangolare (Figura 4.19[26]).

$$\phi_{mid} = \frac{\sum_{i=1}^{n} \phi_i}{n}; \tag{4.26}$$

$$\lambda_{mid} = \frac{\sum_{i=1}^{n} \lambda_i}{n}; \tag{4.27}$$

Le differenze principali, come si può intuire dalla spiegazione dei due metodi, sono essenzialmente due: le performance e l'accuratezza dei risultati. Utilizzando il metodo *Geographic Midpoint* otteniamo una coordinata GPS media molto precisa anche per *POIs* molto distanti fra loro, ma abbiamo anche un numero maggiore di calcoli da effettuare e quindi delle performance peggiori, invece utilizzando l'altro metodo otteniamo delle performance ottimali ma una accuratezza che dipende dalla distanza tra i *POIs* che stiamo

[26]Immagine della mappa creata da *MongoLife*, `https://mongolife.com`.

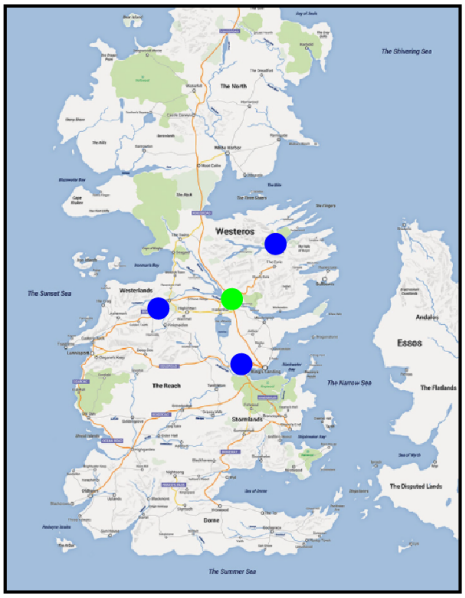

Figura 4.19: Calcolo della posizione GPS di un *Cluster* tramite il metodo *Average Latitude-Longitude*.

considerando: dai risultati ottenuti tramite i test effettuati abbiamo notato che per distanze inferiori ai 300 km l'approssimazione ottenuta è molto simile al risultato ottenuto dal *Geographic Midpoint*, mentre all'aumentare di queste distanze aumenta anche l'errore del risultato. Dato che le possibili applicazioni che possono essere sviluppate utilizzando la libreria *AR POI Experiences* sono molte e potrebbero coprire diversi scenari e diversi ambiti, la scelta tra quale di questi due metodi utilizzare viene lasciata interamente allo sviluppatore in fase di configurazione della scena, in modo tale che possa decidere in modo arbitrario il metodo che ritiene migliore in base alle proprie necessità.

Un piccolo appunto che possiamo fare sull'implementazione di questi due metodi è che il risultato di entrambi viene poi troncato per tenere un massimo di 7 decimali nella latitudine e nella longitudine, questo perchè tutti gli altri

decimali dopo il settimo non ci danno nessuna informazione aggiuntiva che possiamo utilizzare per gli scopi di questa tesi. Per maggiori informazioni sul significato dei decimali utilizzati per la latitudine e la longitudine, quando sono espresse in gradi, facciamo riferimento all'Appendice A.

Una volta calcolata la coordinata GPS media in cui posizionare ogni *Cluster* creato, abbiamo tutte le informazioni necessarie per poter applicare ad ogni *Cluster* la procedura descritta precedentemente per la creazione dei relativi *Renderable*. In Figura 4.20 possiamo vedere i risultati ottenuti dopo aver applicato il *Clustering* alla nostra scena: a sinistra la scena senza *Clustering* con alcuni *Renderable* sovrapposti, mentre a destra abbiamo la stessa scena in cui però è stato abilitato il *Clustering*.

Figura 4.20: A sinistra una scena senza l'utilizzo del *Clustering* con il problema dei *POIs* sovrapposti, mentre a destra applicando il *Clustering* vediamo soltanto il *Renderable* associato a quel *Cluster*.

4.9.2 Aggiornamento dei Cluster

I *Cluster* possiedono tutte le caratteristiche possedute da un *RealPoiNode*:

- hanno un *POI*, che però non viene assegnato direttamente dallo sviluppatore ma viene creato a runtime con i metodi descritti precedentemente;
- è possibile associarci un *Renderable* ed un *ViewRenderable*;

- è possibile definire un comportamento da attivare in caso di *tap* dell'utente oppure uno da eseguire ad ogni ciclo di rendering;

- è possibile configurare l'aggiornamento di un *Renderable* tramite il *Metodo Linear* oppure tramite il *Metodo Teleport* descritti precedentemente.

Una particolare osservazione va fatta sull'ultimo punto, ovvero su come poter effettuare la transizione tra la vecchia posizione di un *Cluster* e la sua nuova posizione nel momento in cui la scena viene aggiornata. Tra due aggiornamenti consecutivi della scena potrebbero essere cambiate molte cose, ad esempio l'utente potrebbe essersi spostato di qualche metro, potrebbe aver cambiato angolazione da cui guardare la scena oppure potrebbe anche solo aver abilitato o disabilitato alcuni contenuti all'interno della scena, per cui non potendo sapere con certezza quali saranno i *Renderable* visibili al prossimo aggiornamento della scena non possiamo nemmeno sapere se al $frame_{i+n}$ i *Cluster* saranno formati dagli stessi *Renderable* rispetto al precedente $frame_i$.

In Figura 4.21 possiamo vedere un esempio grafico di questo problema, ovvero supponendo che il *Clustering* sia attivo, quando la scena si aggiorna ad un certo $frame_i$ l'utente avrà un totale di 6 *POIs* visibili, 3 dei quali formano il *Cluster A* ed altri 2 formano il *Cluster B*. Nel frattempo l'utente esplora la scena e si muove, motivo per cui al prossimo aggiornamento della scena in un certo $frame_{i+n}$ l'utente vedrà sempre gli stessi 6 *POIs* ma da un'angolazione diversa, causando così la creazione di un solo *Cluster C* composto da 4 di questi *POIs*. Come possiamo vedere, in rosso abbiamo cerchiato ed indicato un particolare *POI* che al $frame_i$ si trovava all'interno del *Cluster A*, mentre all'aggiornamento successivo si trovava all'interno del *Cluster C*, che oltretutto è l'unico presente nella scena. A questo punto ci troviamo proprio nella situazione problematica introdotta precedentemente: come facciamo a determinare se la transizione deve essere tra i *Cluster A* e *C* oppure tra i *Cluster B* e *C*?

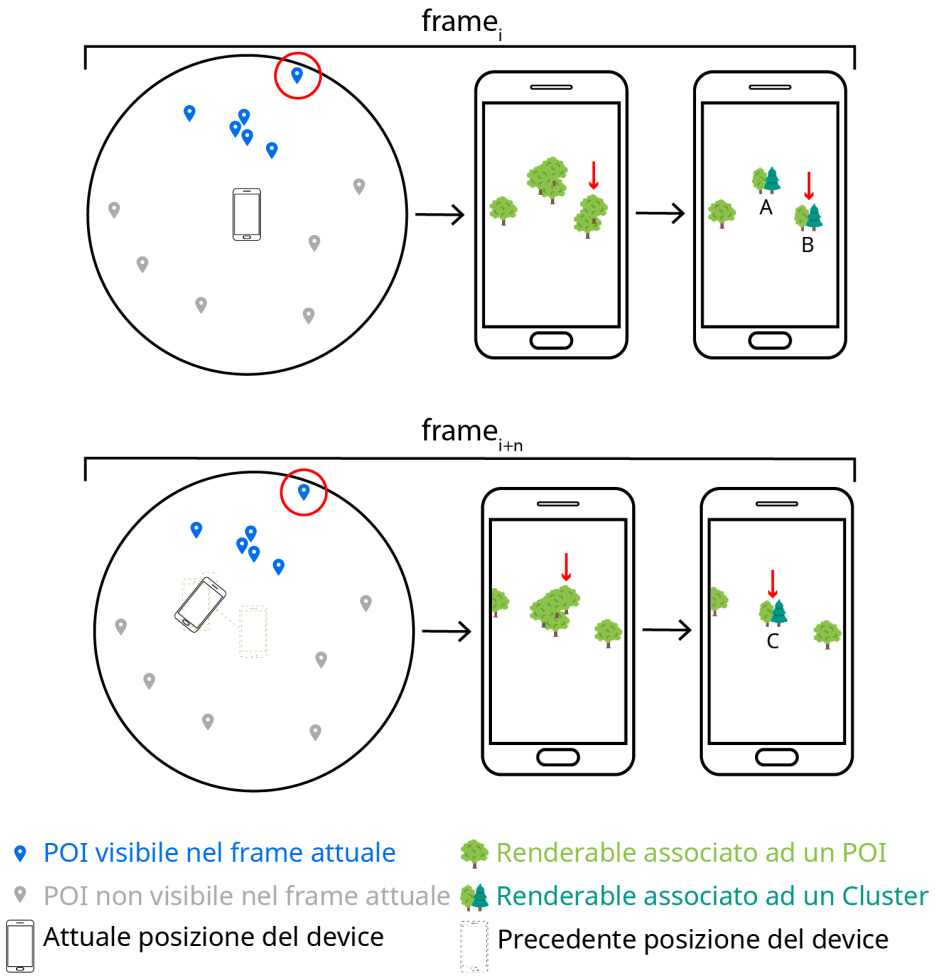

Figura 4.21: Problema legato all'aggiornamento della posizione di un *Cluster* utilizzando la *Modalità Linear*, non possiamo conoscere il punto di partenza corretto della transizione.

Per le ragioni esposte precedentemente, quello che viene fatto ad ogni aggiornamento della scena è applicare la procedura di *detach* a tutti i *Cluster*, in modo tale da rimuoverli dalla scena e dalla memoria del device, per poi effettuare nuovamente i calcoli descritti al fine di determinare quali sono i *Renderable* da renderizzare. Questo però introduce un ulteriore problema: se tra due aggiornamenti consecutivi della scena l'utente non si è spostato, siamo nel caso in cui potenzialmente nella scena al *frame* corrente abbiamo

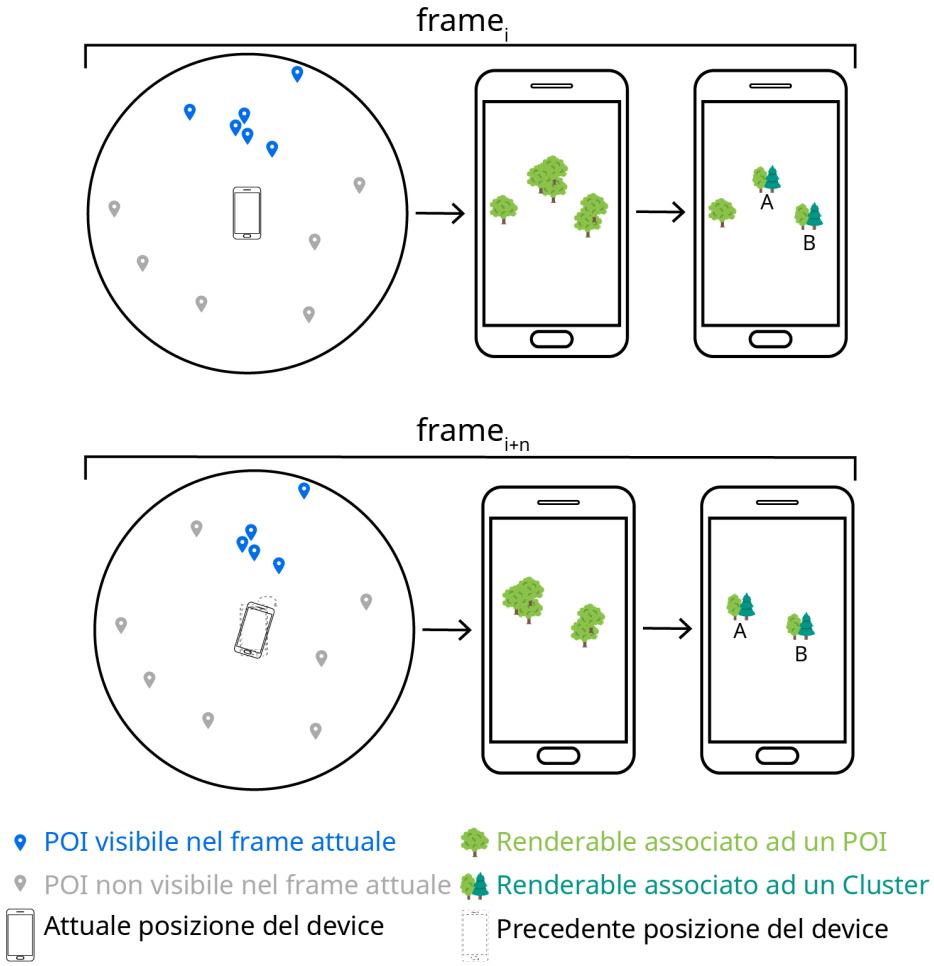

Figura 4.22: Problema legato all'aggiornamento della posizione di un *Cluster* senza utilizzare la *Modalità Linear*. In questo caso tra due aggiornamenti della scena consecutivi possiamo determinare con esattezza le posizioni di partenza e di destinazione dei *Cluster* A e B poiché non sono cambiati.

ricreato un *Cluster* identico a quello creato al *frame* precedente, per cui avrebbe senso mostrare una transizione tra la vecchia e la nuova posizione invece di farlo scomparire tramite il *detach* per poi ricrearlo (Figura 4.22).

Al fine di risolvere questa intricata problematica, nel caso in cui lo sviluppatore abbia abilitato l'aggiornamento dei *Renderable* tramite la *Modalità Linear*, abbiamo scelto di adottare la seguente strategia:

1. invece di effettuare immediatamente il *detach* su tutti i *Cluster* calcolati al *frame* precedente, disabilitiamo il rendering del relativo contenuto virtuale e li memorizziamo in un'apposita lista che per comodità chiameremo *oldClusters*;

2. effettuiamo tutti gli step descritti precedentemente per calcolare i nuovi *Cluster* relativi al *frame* corrente e li memorizziamo in una lista chiamata *newClusters*;

3. ordiniamo le due liste di *Cluster* per distanza crescente dal device;

4. creiamo una transizione tra tutte le coppie di *Cluster* che si trovano nella stessa posizione di lista, per cui avremo una transizione dalla posizione GPS di *oldClusters.get(0)* a quella di *newClusters.get(0)* e così via;

5. alla fine di questi passaggi effettuiamo il *detach* su tutti i *Cluster* contenuti in *oldClusters*.

Questa scelta implementativa ci ha quindi permesso di dare la possibilità allo sviluppatore di poter decidere se aggiornare la scena utilizzando la *Modalità Linear* oppure no anche per quanto riguarda i *Cluster*. Ovviamente siamo consapevoli che utilizzare la distanza come metrica per creare la transizione non è la scelta ottimale, in quanto potrebbe capitare che la transizione avvenga tra due *Cluster* che contengono *POIs* completamente differenti. Determinare una nuova metrica è sicuramente uno dei possibili sviluppi futuri per il miglioramento di questa libreria, per esempio si potrebbero implementare dei confronti in base al grado di similarità tra gli elementi delle liste *oldClusters* e *newClusters* confrontando quali sono i *Cluster* che hanno il maggior numero di *POIs* in comune al loro interno. In ogni caso, abbiamo ritenuto che la soluzione implementata sia una soluzione sufficiente per migliorare l'esperienza dell'utente, il quale, ad esempio, mentre sfrutta un'applicazione creata utilizzando *AR POI Experiences* potrà seguire i movimenti

di tutti i contenuti virtuali nella scena invece di semplicemente vederli sparire improvvisamente per ricomparire in un nuovo punto.

4.9.3 Alternativa al Clustering

Dato che potrebbero esserci alcuni scenari di utilizzo in cui lo sviluppatore non vuole raggruppare i contenuti mostrati nella scena, il *Clustering* non è l'unica soluzione per evitare la sovrapposizione di contenuti virtuali che abbiamo implementato all'interno della libreria *AR POI Experiences*. Una seconda possibile soluzione configurabile dallo sviluppatore è la possibilità di effettuare uno *shift* dei *Renderable* sovrapposti, in modo tale da evitare sia il raggruppamento che la sovrapposizione (Figura 4.23).

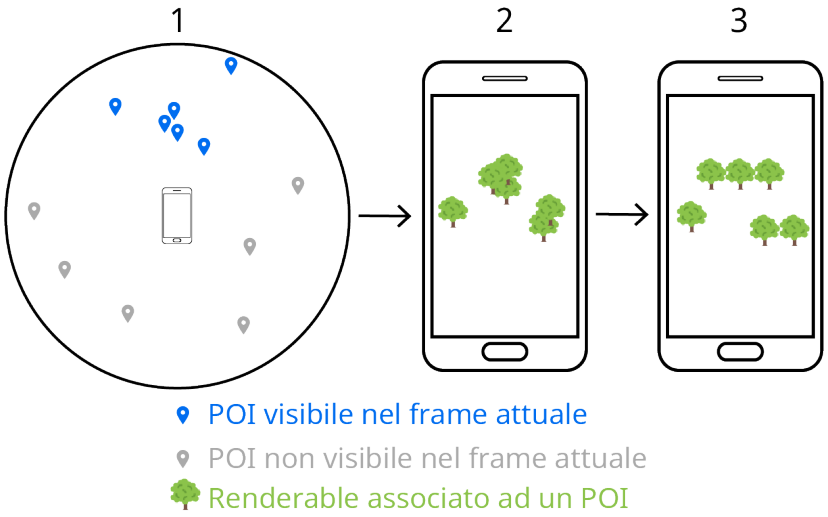

Figura 4.23: Funzionalità per evitare la sovrapposizione dei *Renderable* tramite uno *shift* dei contenuti sovrapposti.

Per scelta implementativa, questa funzionalità non viene configurata a livello di scena ma va configurata singolarmente per ogni *RealPoiNode* che si crea, in modo tale che lo sviluppatore possa decidere arbitrariamente di assegnare diverse direzioni di *shift* ad ogni *POI*. Il controllo che viene fatto per capire se il *Renderable* di un *POI* è sovrapposto ad altri *Renderable*

nella scena è esattamente lo stesso che viene fatto per il *Clustering*, ovvero utilizziamo il metodo *overlapTestAll(Node)* fornito dalle APIs di *ARCore*.

In Figura 4.24 possiamo vedere il risultato di questa funzionalità: a sinistra abbiamo una scena in cui i *Renderable* sono sovrapposti ed il *Clustering* è disabilitato, mentre a destra abbiamo la stessa scena in cui però abbiamo specificato per ogni contenuto di effettuare uno *shift* a destra nel caso in cui sia sovrapposto ad altri *Renderable*.

Figura 4.24: A sinistra una scena con il problema dei *POIs* sovrapposti, mentre a destra la stessa scena in cui abbiamo applicato lo *shift* ai *POIs* sovrapposti.

Dato che il *Clustering* viene abilitato a livello di scena, mentre questa funzionalità viene specificata per ogni singolo *POI*, potrebbe capitare che lo sviluppatore le abbia configurate ed abilitate entrambe. Al fine di evitare calcoli inutili e ripetitivi, abbiamo deciso di dare la precedenza alla funzionalità di *Clustering*, per cui:

- se lo sviluppatore ha abilitato il *Clustering*, allora tutti i *Renderable* sovrapposti finiranno all'interno di un apposito *Cluster* e non verrà fatto nessun controllo aggiuntivo per applicare uno *shift* poiché, dopo il *Clustering*, non ci saranno più contenuti sovrapposti nella scena;

- se invece il *Clustering* è disabilitato e lo *shift* è configurato, allora per ogni *RealPoiNode* in cui è stato configurato andremo ad effettuare

il controllo sulla sovrapposizione per poi applicare il relativo *shift* se necessario;

- se invece nessuna delle due opzioni è abilitata, allora semplicemente i *Renderable* resteranno nelle loro posizione nella scena.

4.10 Gestione POIs Dinamici

Finora abbiamo visto la maggior parte delle funzionalità principali offerte dalla libreria *AR POI Experiences*, tramite le quali possiamo creare delle esperienze in Realtà Aumentata legate a punti di interesse. Ricordandoci i 4 casi descritti in Tabella 2.1, che riportiamo anche qui sotto come Tabella 4.1, tramite le funzionalità descritte finora possiamo coprire tutti i casi in cui i *POIs* sono *Statici* ed il *Tempo* è o *Statico* o *Dinamico*. L'ultima funzionalità

		Tempo	
		Statico	**Dinamico**
POI	**Statico**	La *Location* del *POI* è una sola, è fissa geograficamente ed è persistente nel tempo.	La *Location* del *POI* è una sola ed è fissa geograficamente, ma non è persistente nel tempo.
	Dinamico	Il *POI* ha più di una *Location* e si muove tra esse, ma è persistente nel tempo.	Il *POI* ha più di una *Location* e si muove tra esse e non è persistente nel tempo.

Tabella 4.1: Descrizione dei 4 possibili scenari su cui abbiamo lavorato.

che andiamo a descrivere ora serve proprio per coprire anche gli ultimi due casi, ovvero quelli in cui abbiamo i *POIs Dinamici*.

In fase di analisi su come realizzare questa funzionalità nel modo più corretto abbiamo deciso di svilupparla in una classe a se stante, chiamata *AnimatedNode*, in cui poter gestire la dinamicità dei contenuti virtuali. In

particolare, per poter creare dei *POIs Dinamici* abbiamo bisogno di dare allo sviluppatore la possibilità di inserire due o più *Location* associate ad un solo *POI*.

La creazione di un *POI Dinamico* avviene quindi specificando il *Renderable* da associare al *POI*, una *Location* di partenza ed una di arrivo. Successivamente, tramite l'apposito *Builder Pattern* è possibile specificare come configurare questo *POI*, in particolare è possibile aggiungere altre *Location*. Una volta creato, il *POI Dinamico* può essere aggiunto alla scena in modo tale che il rendering del suo contenuto virtuale venga gestito utilizzando le stesse modalità descritte finora per i *POI Statici*.

Una cosa che è importante sottolineare e definire è il modo in cui viene gestita la dinamicità di questi *POIs*, ovvero il modo in cui un *POI Dinamico* si muove tra le diverse *Location*. Inizialmente volevamo far sì che lo spostamento tra due coordinate GPS associate ad un *POI Dinamico* fosse una vera e propria navigazione stradale, in modo tale che un utente potesse utilizzare il contenuto virtuale come guida lungo il percorso da una *Location A* ad una *Location B*. Per realizzare ciò avremmo dovuto integrare altre APIs e funzionalità di terzi, come ad esempio le *Google Direction APIs*[27], il che avrebbe introdotto nuove problematiche e maggiori complicazioni non necessariamente collegate allo scopo di questa tesi. Per questi motivi e per continuare a seguire la logica di lasciare allo sviluppatore un maggiore libero arbitrio su quali tecnologie utilizzare, abbiamo deciso di implementare il movimento dei *POIs Dinamici* tramite le animazioni fornite dalle APIs di Android, ovvero di utilizzare dei movimenti all'interno dell'ambiente in Realtà Aumentata che potranno essere integrati con funzionalità di navigazione stradale fornite da terzi. In Figura 4.25[28] vediamo nell'immagine di sinistra la classica navigazione stradale tramite l'app *Google Maps*[29], mentre in quella di destra

[27]Google Direction APIs, `https://developers.google.com/maps/documentation/directions/star`.

[28]Icona del cane scaricata dalla piattaforma *Icons8*, `https://icons8.com/icon/set/corgi/all`. Immagine di sfondo presa da *Google Street View*.

[29]Google Maps, `https://www.google.it/maps`.

vediamo quale sarebbe la direzione del movimento applicato sul *Renderable*, il quale si muoverebbe infatti in linea d'aria tra le due *Location*.

Figura 4.25: Gestione dei *POIs Dinamici* all'interno della libreria *AR POI Experiences*. A sinistra una classica navigazione tramite l'applicazione *Google Maps*, mentre a destra vediamo quale sarebbe il movimento in linea d'aria applicato sul *Renderable* per andare dalla *Location A* alla *Location B*.

Nel caso in cui uno sviluppatore volesse utilizzare la libreria *AR POI Experiences* per creare un'applicazione di navigazione stradale in AR per pedoni potrebbe integrarci, ad esempio, le sopracitate *Google Direction APIs* per ottenere le indicazioni stradali da una particolare *Location A* ad una *Location B*, potendo così sfruttare la classe *AnimatedNode* per creare animazioni, di durata arbitraria, che conducano l'utente fino a destinazione passo dopo passo. In Figura 4.26[30] possiamo vedere quale sarebbe un possibile risultato di un'applicazione realizzata integrando queste diverse tecnologie.

[30]Immagini di sfondo prese da *Google Street View*.

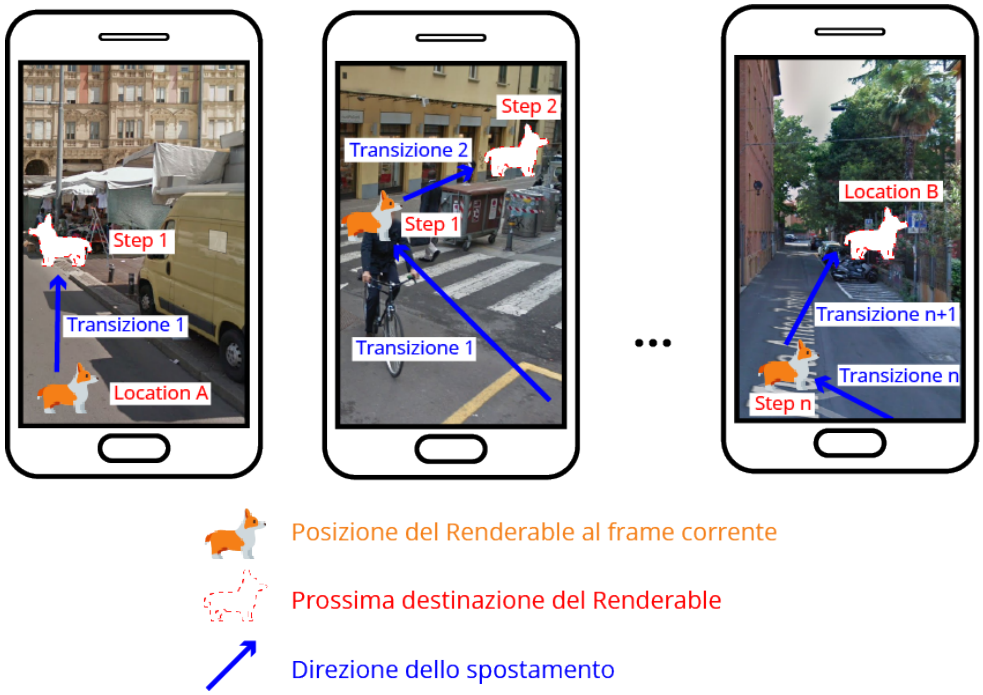

Figura 4.26: Esempio di possibile integrazione tra la gestione dei *POIs Dinamici* e librerie di terze parti per la navigazione stradale. Da sinistra verso destra possiamo vedere quali potrebbero essere gli spostamenti applicabili sul *Renderable* utilizzando le informazioni sulla direzione da seguire.

In generale possiamo quindi concludere affermando che utilizzando la libreria *AR POI Experiences* è possibile anche realizzare applicazioni mobile relative alla creazione di esperienze in Realtà Aumentata legate a *POIs Dinamici*, tenendo in considerazione che a seconda delle diverse esigenze lo sviluppatore può integrare alle funzionalità offerte anche altre librerie ed APIs di terze parti.

Capitolo 5

Case study

5.1 Prototipo sviluppato

Al fine di mostrare quali sono le potenzialità della libreria *AR POI Experiences* che abbiamo creato come prodotto principale della tesi e che abbiamo ampiamente descritto nel corso del capitolo precedente, abbiamo deciso di sviluppare il prototipo di un'app mobile per Android che copra una delle 4 problematiche analizzate. L'idea iniziale era quella di realizzare un prototipo inerente alla problematica di avere *POIs Dinamici* e *Tempo Dinamico*, ad esempio una guida turistica in Realtà Aumentata per mostrare all'utente i monumenti di una città, ma, come descritto nel capitolo precedente, per realizzare applicazioni di questo tipo in modo ottimale serve per forza integrare APIs di terzi che forniscano le informazioni necessarie alla navigazione stradale, il che avrebbe reso lo sviluppo del prototipo molto più complesso ed avrebbe richiesto molto più tempo.

Per evitare questa complicazione non necessaria ai fini dimostrativi, abbiamo deciso di realizzare un prototipo che copra il caso di *POIs Statici* con *Tempo Dinamico*: l'applicazione sviluppata si chiama **WAY**, acronimo di *"Where ARe You?"*, e permette agli utenti di incontrarsi seguendo un indicatore in Realtà Aumentata. In questo caso i *POIs* saranno quindi rappresentati dalle posizioni GPS degli utenti ed il *Tempo* è *Dinamico* poiché il

POI viene rimosso una volta raggiunta la posizione indicata. Come accennato poco fa, dato che siamo nel caso di *POIs Statici*, non abbiamo una vera e propria navigazione stradale ma avremo un contenuto virtuale posizionato in corrispondenza del contatto che si vuole raggiungere. Ad esempio, se prendiamo il caso in cui un utente A voglia raggiungere un utente B, allora A inquadrando il mondo con la fotocamera del suo device potrà sfruttare l'app *WAY* per vedere un indicatore virtuale posizionato proprio sulla coordinata GPS di A.

Le funzionalità implementate in questo prototipo sono le seguenti:

1. aggiunta, modifica o rimozione di un utente dalla propria lista di contatti;

2. possibilità di abilitare o disabilitare la condivisione della propria posizione GPS con uno o più contatti;

3. possibilità di cliccare su uno dei contatti per poterlo cercare tramite un indicatore nella scena in Realtà Aumentata.

Di seguito andremo a spiegare come abbiamo creato il progetto per lo sviluppo di questo prototipo e come abbiamo implementato queste funzionalità tramite l'utilizzo delle librerie realizzate fornendo anche alcune righe di codice come esempio.

5.2 Configurazione del progetto

Come per tutti i progetti Android, la prima cosa da fare è configurare il *Manifest*, il file *build.gradle* ed importare tutte le librerie di cui abbiamo bisogno. Per quanto riguarda il *Manifest*, a differenza di quelli descritti in precedenza per le 3 librerie sviluppate, dobbiamo specificare anche tutte le *Activity* che faranno parte del nostro progetto, che nel nostro caso sono 5:

1. **LoginActivity**, prima pagina visualizzata all'apertura dell'applicazione, si occupa della fase di login e registrazione degli utenti;

2. **MainActivity**, pagina principale visualizzata da tutti gli utenti che hanno già effettuato il login;

3. **ShareActivity**, utilizzata per abilitare o disabilitare la condivisione della propria posizione GPS con i contatti aggiunti;

4. **FindActivity**, molto simile alla *ShareActivity*, mostra la lista dei contatti aggiunti indicando quali hanno condiviso la propria posizione con noi e quali no;

5. **ArActivity**, schermata che si occupa della scena in Realtà Aumentata.

Oltre alla dichiarazione delle *Activity* non dobbiamo dimenticarci dei permessi da specificare, in particolare, in aggiunta ai permessi richiesti dalle librerie che importeremo, andiamo ad aggiungere anche i permessi per poter ottenere le informazioni sui contatti direttamente dalla rubrica del device dell'utente.

Passiamo ora alla configurazione del file *build.gradle* in cui specifichiamo che il livello minimo di *Android APIs* è il *24*[1] ed indichiamo le librerie che vogliamo importare:

- le 3 librerie sviluppate *AR POI Experiences*, *Device Position* e *Device Orientation*;
- i servizi di localizzazione di Google forniti dalle *Google Play Services Location APIs*[2];
- gli SDK per la Realtà Aumentata *ARCore* e *Sceneform*;
- *Firebase*[3] per la condivisione della posizione GPS tra utenti e *Gson*[4] di Google per fare l'upload ed il download delle informazioni in formato *JSON*[5];

[1]Ricordiamo che il *24* è lo stesso livello di *Android APIs* minimo richiesto dalle tre librerie sviluppate e descritte nei capitoli precedenti.

[2]Google APIs: Play Services Location, `https://developers.google.com/android/reference/com/google/android/gms/location/package-summary`.

[3]Firebase, `https://firebase.google.com`.

[4]Google Gson, `https://github.com/google/gson`.

[5]JSON, `https://www.json.org/`.

- infine le librerie *Sweet Alert Dialog*[6] e *Swipe Menu ListView*[7] che forniscono la possibilità di creare, rispettivamente, componenti *Dialog*[8] e componenti *ListView*[9] con qualche funzionalità aggiuntiva.

5.3 Progettazione del prototipo

Dato che il prototipo è stato realizzato per scopo puramente dimostrativo e non è il prodotto principale di questa tesi, non sono state implementate tutte le funzionalità, in particolare tutta la parte relativa alla gestione degli utenti e della privacy non è stata presa in esame. Infatti, l'*Activity* che si occupa del login degli utenti è stata realizzata puramente a livello grafico: non memorizza nessuna informazione degli utenti e non ha nessun controllo sulla correttezza dei dati inseriti. Ciò nonostante, come vedremo nel corso di questo capitolo, la condivisione della posizione GPS di un device è stata implementata in modo tale che potesse funzionare realmente tra device diversi per poter simulare una comunicazione in tempo reale tra gli utenti.

Di seguito mostriamo come sono state realizzate le varie *Activity* per il prototipo, mentre nel prossimo sottocapitolo andiamo a spiegare come sono state implementate le funzionalità descritte.

5.3.1 Login

Come anticipato, il login funziona puramente a livello grafico. In Figura 5.1[10] possiamo vedere che l'*Activity* è composta da un form per inserire le

[6]Sweet Alert Dialog, `https://github.com/thomper/sweet-alert-dialog`.

[7]Swipe Menu ListView, `https://github.com/baoyongzhang/SwipeMenuListView`.

[8]Android APIs: Dialog, `https://developer.android.com/guide/topics/ui/dialogs`.

[9]Android APIs: ListView, `https://developer.android.com/guide/topics/ui/layout/listview`.

[10]Icona utilizzata come logo del prototipo realizzata da *Freepik*, `https://www.freepik.com/`. Scaricata dalla piattaforma *FlatIcon*, `https://www.flaticon.com/`.

credenziali di accesso e da un bottone utilizzato per passare alla schermata successiva.

Figura 5.1: Prototipo *WAY*: pagina di login.

5.3.2 Main

La pagina principale dell'applicazione è composta semplicemente dalle due possibili azioni che l'utente può eseguire, ovvero condividere la propria posizione GPS con i propri contatti oppure cercare uno dei contatti aggiunti sfruttando la Realtà Aumentata (Figura 5.2).

5.3.3 Share

In questa *Activity* abbiamo una semplice *ListView* che mostra quali sono gli utenti che abbiamo aggiunto alla nostra lista di contatti e mostra se al momento abbiamo condiviso la posizione con loro oppure no (Figura 5.3).

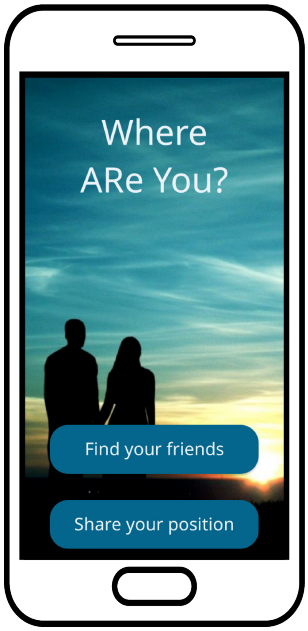

Figura 5.2: Prototipo *WAY*: pagina principale visualizzata dopo il login.

Per aggiungere gli utenti alla nostra lista di contatti ci basta cliccare sul *FAB*[11] in basso a destra.

Come possiamo vedere in Figura 5.4 è possibile modificare o rimuovere un contatto dalla lista tramite uno *swipe*[12] da destra verso sinistra, mentre in Figura 5.5 possiamo vedere che è possibile utilizzare il widget *Switch*[13] in alto per abilitare o disabilitare la condivisione della propria posizione GPS con tutti i contatti della lista.

[11]L'acronimo *FAB* sta per *Floating Action Button*, un bottone di forma circolare utilizzato per avviare l'azione principale all'interno di una *UI*. Nel nostro caso, l'azione associata al *FAB* è quella di aggiunta di un contatto.

[12]Lo *swipe* è una delle possibili *gesture* che si possono utilizzare su alcuni dispositivi, in particolare consiste nell'azione di far scorrere il dito lateralmente, nel nostro caso in direzione da destra verso sinistra.

[13]Android Widgets: Switch, `https://developer.android.com/reference/android/widget/Switch`.

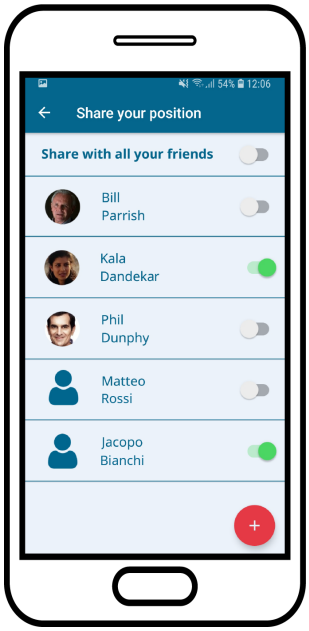

Figura 5.3: Prototipo *WAY*: condivisione della propria posizione GPS con i contatti aggiunti. Gli *Switch* verdi identificano i contatti con cui la stiamo condividendo.

5.3.4 Find

Questa pagina è molto simile a quella descritta precedentemente, dove al posto dello *Switch* abbiamo un'icona che per ogni contatto ci indica se al momento quel contatto sta condividendo la posizione con noi oppure no (Figura 5.6). Nel caso in cui la stia condividendo con noi, allora cliccando sul contatto possiamo avviare l'*Activity* che si occupa della scena in Realtà Aumentata e possiamo iniziare a seguire il contenuto virtuale fino a destinazione. Anche in questa *Activity* possiamo aggiungere i contatti utilizzando l'apposito *FAB* in basso a destra.

5.3.5 Scena in AR

L'ultima *Activity* è proprio quella che si occupa di visualizzare la scena in Realtà Aumentata. Come possiamo vedere in Figura 5.7, all'interno del-

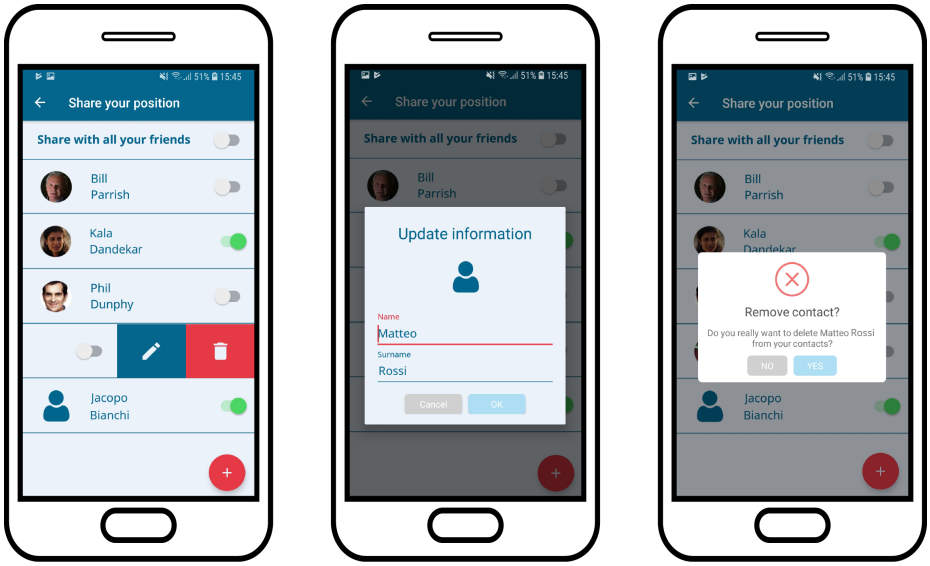

Figura 5.4: Prototipo *WAY*, da sinistra verso destra abbiamo le seguenti schermate: swipe da destra a sinistra sul contatto, click sul tasto blu per modificare le informazioni del contatto e click sul tasto rosso per rimuoverlo dalla lista.

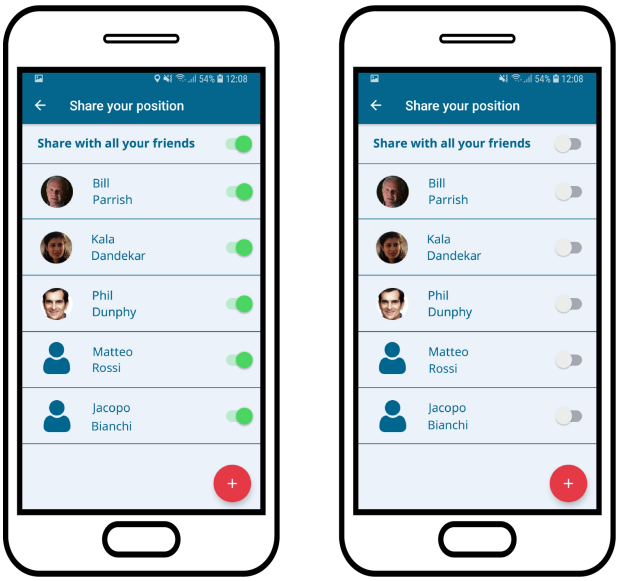

Figura 5.5: Prototipo *WAY*: *Switch* per abilitare o disabilitare la condivisione della posizione con tutti i contatti.

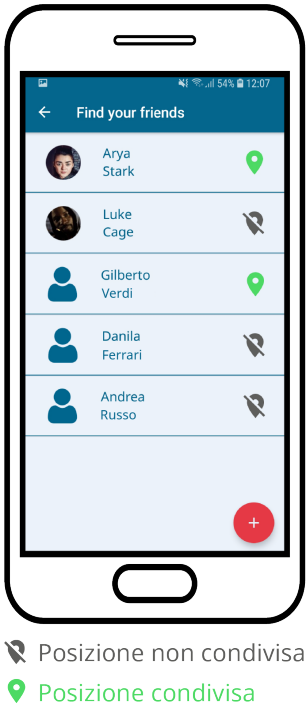

Figura 5.6: Prototipo *WAY*: pagina da cui poter iniziare la ricerca in Realtà Aumentata di un contatto.

la scena avremo il contenuto virtuale che ci indica la posizione del contatto che stiamo cercando. Il contenuto virtuale è composto da una freccia 3D[14] utilizzata come *Renderable* e da una *TextView*[15] utilizzata come *ViewRenderable* nella quale vediamo la distanza in metri che ci separa dalla nostra destinazione.

Sempre nella Figura 5.7 possiamo anche notare un secondo *Renderable*[16]

[14]Il modello 3D di partenza è stato realizzato da Patrick Coffey ed era caricato sulla piattaforma *Poly* di Google: `https://poly.google.com/view/8xde5IVnRR_`. Il modello è stato poi modificato per adattarlo alle nostre esigenze utilizzando il software *MeshLab*: `http://www.meshlab.net`.

[15]Android Widgets: TextView, `https://developer.android.com/reference/android/widget/TextView`.

[16]Questo contenuto 3D è stato realizzato da Mark Steelman ed era caricato sulla piattaforma *Poly* di Google: `https://poly.google.com/view/43TYiGXodkM`. Anche questo modello è stato poi modificato per adattarlo alle nostre esigenze utilizzando *MeshLab*.

Figura 5.7: Prototipo *WAY*: esempio di scena in Realtà Aumentata. In alto vediamo una freccia verde che è il *Renderable* associato al *POI* dell'utente che stiamo cercando, mentre in basso abbiamo un secondo *Renderable* che ci guida e ci indica la direzione in cui si trova l'utente.

posizionato in basso che indica in quale direzione si trova il *POI* che stiamo cercando. Questo contenuto è stato creato all'interno dell'*Activity* utilizzando solamente le primitive fornite da *ARCore* e *Sceneform* ed è stato realizzato per aiutare l'utente nella ricerca del contatto selezionato. Per realizzarlo abbiamo sfruttato le informazioni sulle posizioni in AR della fotocamera e dell'*Anchor* legata al *POI* per calcolare l'angolo tra i due.

5.4 Aggiunta di un contatto

L'azione di aggiungere un utente alla propria lista di contatti all'interno di una generica applicazione mobile può essere sviluppata in diversi modi: invito all'interno dell'app, via mail, tramite messaggio e così via. Nel nostro caso abbiamo scelto di implementare l'aggiunta di un utente utilizzando i contatti

memorizzati all'interno della rubrica del device. Tale aggiunta dovrebbe quindi avere le seguenti caratteristiche:

1. possibilità di scegliere un contatto dalla propria rubrica del device;
2. invio di una richiesta di aggiunta al contatto selezionato;
3. attesa di una sua conferma prima di poterlo aggiungere effettivamente alla lista di contatti.

Dato che non abbiamo una reale gestione degli utenti implementata nel prototipo poiché non necessaria per la dimostrazione delle funzionalità della libreria *AR POI Experiences*, l'aggiunta di un contatto è stata implementata senza il controllo descritto al punto 2: basta selezionare uno dei propri contatti dalla rubrica e questo verrà aggiunto immediatamente in modo fittizio alla propria lista di contatti all'interno dell'applicazione.

Per implementare l'aggiunta di un contatto all'interno delle *Activity Share* e *Find* abbiamo interrogato direttamente la rubrica del device sfruttando un *Intent*[17] di tipo *ACTION_PICK*[18], in modo tale da ottenere le informazioni del contatto selezionato dall'utente all'interno della propria rubrica. Per ognuna delle informazioni ottenute abbiamo poi creato un apposito spazio all'interno della lista sfruttando un **ArrayAdapter**[19], ovvero uno strumento che ci permette di creare un componente per ogni diverso oggetto contenuto all'interno dei dati che gli passiamo. Nel nostro caso i dati che passiamo a questo *ArrayAdapter* saranno ovviamente la lista di contatti e, per ognuno di essi, andiamo a creare e popolare il *layout* utilizzato all'interno della *ListView*.

In Figura 5.8 possiamo vedere i passi che l'utente può eseguire per aggiungere un contatto dalla propria rubrica: per prima cosa si clicca sul *FAB*

[17]In Android un *Intent* è una descrizione astratta di una particolare operazione che vogliamo eseguire. Nel nostro caso l'azione che volevamo eseguire è quella di ottenere le informazioni di un contatto selezionandolo dalla rubrica del device.

[18]Android Intents: ACTION_PICK, `https://developer.android.com/reference/android/content/Intent#ACTION_PICK`.

[19]Android APIs: ArrayAdapter, `https://developer.android.com/reference/android/widget/ArrayAdapter`.

in basso a destra, si seleziona il contatto, successivamente si possono ricontrollare le informazioni sul contatto ed infine lo si aggiunge a tutti gli effetti alla lista contatti all'interno dell'app.

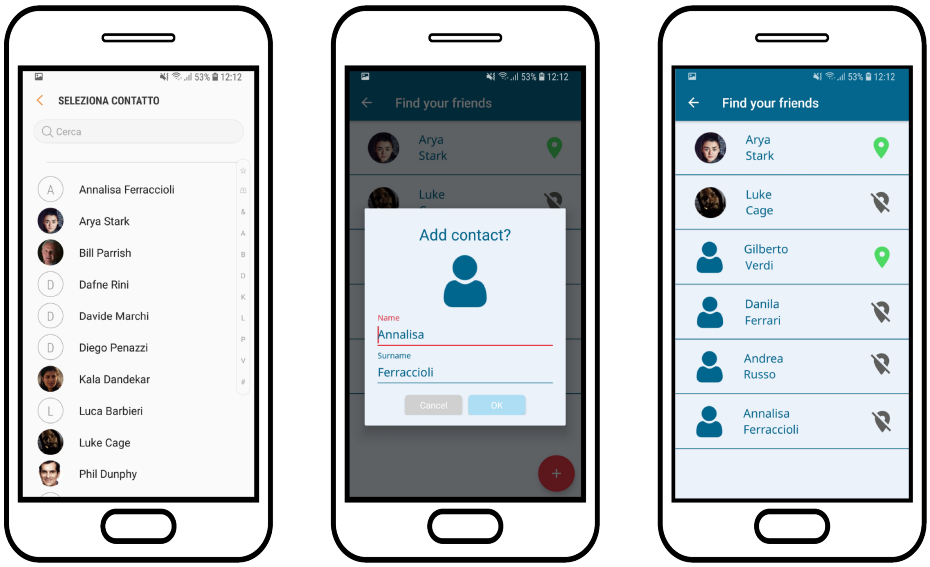

Figura 5.8: Prototipo *WAY*: procedura di aggiunta di un contatto. Dopo aver cliccato il *FAB* per aggiungere un contatto abbiamo, da sinistra verso destra: selezione di un contatto dalla rubrica, controllo delle informazioni del contatto ed infine l'aggiunta del contatto in fondo alla lista.

5.5 Condivisione della posizione GPS

Altra funzionalità implementata correttamente, utile per effettuare qualche test con lo scambio di informazioni tra due device, è stata la condivisione della posizione GPS tra due utenti. Come accennato in precedenza, per realizzarla abbiamo sfruttato le APIs di *Firebase*, in particolare il **Firebase Realtime Database**[20] che permette di memorizzare dati in formato *JSON* all'interno di un cloud e di sincronizzarli con tutti i device connessi.

[20]Firebase Realtime Database, `https://firebase.google.com/docs/database/`.

Abbiamo scelto di utilizzare *Firebase* per l'implementazione di questa funzionalità principalmente per due motivazioni:

1. grazie alla sincronizzazione automatica, ogni volta che un dato all'interno del *Firebase Realtime Database* viene modificato anche i dati posseduti dai device connessi vengono automaticamente aggiornati;

2. tutti i dati vengono memorizzati automaticamente anche in locale, in modo tale che nel caso in cui la connessione Internet smetta di funzionare è comunque possibile accedere all'ultimo dato ricevuto e continuare ad utilizzare l'applicazione in modalità offline.

All'interno del *Firebase Realtime Database* è inoltre possibile definire e specificare delle regole per poter accedere ai dati affinché ogni utente possa accedere soltanto ai dati di sua competenza. Nel nostro particolare caso, dato che non ci siamo occupati della parte relativa alla gestione ed alla sicurezza degli utenti, non abbiamo implementato nessuna regola particolare, motivo per cui qualunque utente in possesso del prototipo accede agli stessi dati.

Per quanto riguarda la parte implementativa su Android, per la condivisione della posizione GPS abbiamo realizzato un *Service*[21] che si occupa di inviare periodicamente i dati sulla posizione GPS dell'utente al *Firebase Realtime Database*. Il *Service* è stato realizzato in modo tale per cui tutti i dati, prima di essere inviati, vengono convertiti in formato *JSON* utilizzando le APIs messe a disposizione da *Gson* di Google. Inoltre, per evitare che l'utente debba tenere aperta l'applicazione, abbiamo fatto in modo che il *Service*, nel momento in cui si accorge che l'applicazione è stata chiusa, si promuova automaticamente a *Foreground Service*[22] per poter continuare le operazioni di condivisione. Per far sì che l'utente sappia che sta condividendo la propria posizione GPS con altri utenti gli inviamo una notifica ad ogni aggiornamento della posizione condivisa.

[21] Android Services, `https://developer.android.com/guide/components/services`.

[22] In accordo con la documentazione Android, con *Foreground Service* si intende un servizio che esegue operazioni evidenti all'utente, ovvero operazioni di cui l'utente è consapevole.

Quindi, grazie all'utilizzo di *Firebase* e di un *Foreground Service*, siamo stati in grado di condividere realmente la posizione GPS tra due utenti che avevano il prototipo installato sui loro device.

5.6 Creazione della scena in AR

Passiamo ora alla funzionalità principale per cui abbiamo realizzato questo prototipo: dimostrare come poter creare un'esperienza in Realtà Aumentata legata a punti di interesse utilizzando la libreria *AR POI Experiences* e quali sono i possibili vantaggi che derivano dal suo utilizzo. Di seguito mostreremo passo per passo tutte le configurazioni necessarie che abbiamo implementato per lo sviluppo del prototipo descritto, in modo tale da fornire anche una piccola guida ed una dimostrazione della semplicità di utilizzo della libreria.

5.6.1 Layout

La prima cosa da fare è la creazione del *layout* che conterrà la nostra scena in AR. Per crearlo abbiamo utilizzato un *ArFragment*[23] fornito da *Sceneform* che contiene già tutto il necessario per poter interagire correttamente con l'ambiente in Realtà Aumentata che andremo a creare.

```
1 <fragment
2   android:name="com.google.ar.sceneform.ux.ArFragment"
3   android:id="@+id/fragment_ar_activity"
4   android:layout_width="match_parent"
5   android:layout_height="match_parent" />
```

In particolare, come descritto nel sottocapitolo 2.5.6, questo *ArFragment* controlla automaticamente se nel device è stata installata la versione corretta di *ARCore* e *Sceneform* e gestisce in modo automatico il controllo a runtime

[23]Sceneform APIs: ArFragment, `https://developers.google.com/ar/reference/java/com/google/ar/sceneform/ux/ArFragment`.

dei permessi necessari per la creazione della scena in AR, ad esempio l'accesso alla fotocamera.

5.6.2 Fase di creazione dell'Activity

Passiamo ora a tutte le operazioni che vanno fatte nel momento in cui l'*Activity* viene creata, operazioni che di conseguenza specificheremo all'interno del metodo *onCreate*[24]. Iniziamo con l'avviare le librerie *Device Position* e *Device Orientation* per ottenere tutte le informazioni necessarie. Per quanto riguarda il parametro di tipo *Device Position* dobbiamo specificare l'*Activity* ed il *Context*[25] in cui ci troviamo, un intero utilizzato per controllare e richiedere a runtime i permessi necessari per attivare la localizzazione dell'utente ed infine una *LocationCallback*[26] in cui indichiamo che, ogni volta che riceviamo una posizione GPS nuova, vogliamo controllare se è migliore o peggiore della precedente sfruttando il metodo *checkAndSetCurrentBestLocation(Location)* descritto nel Capitolo 3.3.

```
1 mDevicePosition = new DevicePosition(this, this, 10,
2   new LocationCallback() {
3     @Override
4     public void onLocationResult(LocationResult lr) {
5       mDevicePosition.checkAndSetCurrentBestLocation(lr);
6     }
7   }
8 );
```

Invece nel caso del parametro di tipo *Device Orientation* dobbiamo indicare l'*Activity* in cui ci troviamo e specificare che vogliamo rimappare il

[24]Android APIs: Activity onCreate, `https://developer.android.com/reference/android/app/Activity.html#onCreate(android.os.Bundle)`.

[25]Ricordiamo che su Android quando si parla di *Context* ci si riferisce all'oggetto che contiene tutte le informazioni sull'ambiente dell'applicazione, utile per accedere a particolari risorse, ricevere *Intents* oppure lanciare *Activity*.

[26]Per maggiori informazioni sulla *LocationCallback*, consultare Capitolo 3.3.

sistema di coordinate[27] utilizzato dai sensori per farlo combaciare a quello utilizzato da *ARCore* per l'ambiente in Realtà Aumentata.

```
mDeviceOrientation = new DeviceOrientation(
    this, DeviceOrientation.X_AXIS, DeviceOrientation.Z_AXIS
);
```

Successivamente dobbiamo ricordarci di creare un *listener*[28] per ricevere gli aggiornamenti sui dati della posizione GPS condivisa contenuti sul *Firebase Realtime Database* ed un secondo *listener*[29] da associare alla scena in Realtà Aumentata creata tramite *ARCore*.

Infine non dobbiamo dimenticarci di creare i *ModelRenderable*[30], ovvero i modelli 3D che saranno poi utilizzati dai singoli *POIs* per la creazione dei singoli *Renderable* da renderizzare all'interno della scena. Maggiori informazioni su come poterli creare vengono date in Appendice B.

5.6.3 ARCore listener

Ora che abbiamo configurato correttamente tutti i parametri necessari, passiamo alle configurazioni principali che implementiamo all'interno del metodo associato come *listener* alla scena in AR. Una cosa importante da specificare è che tutto ciò che definiamo all'interno di questo *listener* verrà eseguito ad ogni *frame*, motivo per cui dobbiamo inserire soltanto lo stretto necessario alla creazione della scena, ma allo stesso tempo è il punto ideale in

[27] Per maggiori dettagli sul perché dobbiamo rimappare il sistema di coordinate fare riferimento al Capitolo 3.2.

[28] Firebase Realtime Database: Read and Write Data on Android, `https://firebase.google.com/docs/database/android/read-and-write`.

[29] ARCore APIs: Scene addOnUpdateListener, `https://developers.google.com/ar/reference/java/com/google/ar/sceneform/Scene#addOnUpdateListener(com.google.ar.sceneform.Scene.OnUpdateListener)`.

[30] Sceneform APIs: ModelRenderable, `https://developers.google.com/ar/reference/java/com/google/ar/sceneform/rendering/ModelRenderable`.

cui poter inserire le azioni da eseguire una volta che la scena è stata creata, come ad esempio processare il *frame* corrente.

Dato che questo metodo viene eseguito ad ogni *frame*, la prima cosa da fare è controllare se la scena è già stata creata in un qualche *frame* precedente oppure no: in caso negativo, procediamo alla creazione ed alla configurazione dei parametri utilizzando l'apposito *Builder Pattern* descritto nei capitoli precedenti.

```
1 /* Creiamo della scena tramite il costruttore. */
2 mArPoiScene = new ArPoiScene(mArFragment.getArSceneView());
3 /* Configuriamo la scena usando il Builder Pattern. */
4 mArPoiScene.configure(s -> s
5     .setRefreshWithTimer(true)
6     .setSceneRefreshTimer(5000) // millisecondi
7     .setFovClippingEnabled(false)
8     .setFarPlaneClippingEnabled(false)
9     .setNearPlaneClippingEnabled(false)
10    .disablePoiClustering()
11    .setPoiUpdateMethod(ArPoiScene.PoiUpdateMethods.LINEAR)
12 );
```

Andiamo ora ad analizzare come è stata configurata la scena:

- abbiamo abilitato il *Refresh Timer* specificando che vogliamo forzare l'aggiornamento della scena ogni 5000 millisecondi, ovvero ogni 5 secondi;

- abbiamo disabilitato tutte e 3 le tipologie di *Clipping* perché all'interno del prototipo avremo sempre un solo *Renderable* legato alla posizione GPS della persona che stiamo cercando;

- per lo stesso motivo esposto sopra, abbiamo disabilitato anche il *Clustering*;

- infine abbiamo impostato la *Modalità Linear*[31] come metodo di aggiornamento dei *POIs*.

Quindi, come abbiamo visto, la creazione e la configurazione della scena in Realtà Aumentata sono operazioni molto semplici e bastano pochissime righe di codice.

Ora procediamo alla creazione del *RealPoiNode* relativo alla coordinata GPS dell'utente che stiamo cercando. Per prima cosa istanziamo l'oggetto sfruttando il costruttore nel quale dobbiamo specificare la coordinata GPS ed il *Node*[32] che verrà utilizzato per la fase di rendering.

```
/* Creiamo il POI tramite il suo costruttore. */
RealPoiNode realPoiNode = new RealPoiNode(
    mPoiData.getLatitude(), mPoiData.getLongitude(),
    mPoiData.getAltitude(), createNode(mModelRenderable)
);
```

Subito dopo la creazione passiamo alla configurazione del *RealPoiNode*.

```
/* Configuriamo il POI usando il Builder Pattern. */
realPoiNode.configure(r -> r
    .setName(mPoiData.getName())
    .setScalingType(
        RealPoiNode.ScalingType.LINEAR_SCALING_WITH_DISTANCE
    )
    .setViewRenderableNode(viewRenderableNode)
    .setViewRenderableNodePosition(
        RealPoiNode.ViewRenderableNodePosition.TOP
    )
    .setRenderEvent(() -> myRenderEvent())
    .setOnTapEvent(() -> myOnTapEvent())
    .setOverlapSettings(RealPoiNode.OverlapSettings.IGNORE)
);
```

[31] Ricordiamo che la *Modalità Linear* è il metodo utilizzato per creare una transizione tra la vecchia posizione del *POI* e la nuova posizione calcolata al momento dell'aggiornamento della scena.

[32] La creazione di un *Node* è arbitraria e dipende da come lo sviluppatore vuole crearlo e configurarlo.

```
15 /* Aggiungiamo il POI alla scena. */
16 mArPoiScene.addRealPoiNode(realPoiNode);
```

Come fatto sopra per la configurazione della scena, andiamo ad analizzare e descrivere la configurazione applicata a questo *POI*:

- gli abbiamo assegnato un nome;
- abbiamo configurato uno *scaling* con la distanza, ovvero all'aumentare della distanza tra il device ed il *POI* il *Renderable* sarà rimpicciolito per dare un'illusione di lontananza;
- abbiamo assegnato un *ViewRenderable* a questo *POI*, il quale conterrà la distanza in metri tra il device e l'utente cercato e sarà posizionato sopra al *Renderable*;
- abbiamo associato un *renderEvent* ed un *onTapEvent* che saranno utili per, rispettivamente, aggiornare la distanza in metri tra device e *POI* visualizzata nel *ViewRenderable* e per far apparire o scomparire dalla scena il *ViewRenderable*;
- dato che ci sarà un solo *POI* nella scena, non abilitiamo lo *shift*;
- infine abbiamo aggiunto il *RealPoiNode* alla scena, in modo tale che venga utilizzato all'interno del ciclo di rendering.

Anche in questo caso possiamo notare come la creazione e la configurazione di un *POI* sia semplice e necessiti di poche righe di codice.

Una volta che abbiamo configurato la scena ed il *POI*, operazione che va fatta solo la prima volta, l'ultima cosa che ci resta da fare è processare il *frame* corrente per renderizzare i contenuti virtuali associati al *POI*. Per farlo utilizziamo il metodo *processFrame* che abbiamo descritto nel Capitolo 4.3 a cui dobbiamo passare il *frame* corrente, la posizione GPS del device ed il *Bearing* del device, tutte informazioni che otteniamo dalle librerie *Device Position* e *Device Orientation*.

```
1 /* Processiamo il frame attuale. */
2 mArPoiScene.processFrame(frame,
3     mDevicePosition.getCurrentBestLocation(),
4     mDeviceOrientation.getRoundedBearing()
5 );
```

In conclusione, come abbiamo visto grazie a questo esempio pratico, possiamo affermare che utilizzare la libreria *AR POI Experiences* per la creazione di un prototipo è un'operazione molto semplice che richiede poche righe di codice sia per la configurazione della scena che per quella dei contenuti associati a punti di interesse, fornendo allo stesso tempo un valido strumento per la creazione di esperienze in Realtà Aumentata legate a punti di interesse.

Conclusioni

Nel corso di questo scritto abbiamo visto come, a partire da un bisogno iniziale specifico di un'azienda, è stato possibile realizzare un prodotto che si propone come possibile soluzione per una problematica più ampia e generica sfruttando la Realtà Aumentata, tecnologia sempre più emergente e popolare che abbiamo potuto analizzare nel dettaglio a partire dalle sue origini fino ad oggi. Come descritto nel corso dei capitoli precedenti, le tecnologie che ci permettono di realizzare prodotti in AR sono tantissime ed in continuo aumento, ognuna delle quali ha i suoi pregi ed i suoi difetti a seconda delle nostre necessità: nel nostro caso abbiamo scelto di utilizzare *ARCore* e *Sceneform* forniti da Google, strumenti grazie al quale siamo riusciti a realizzare la libreria Android *AR POI Experiences* proposta come possibile soluzione per facilitare la creazione di esperienze in Realtà Aumentata legate a punti di interesse.

Per sviluppare questa possibile soluzione abbiamo seguito un processo semplificato di *User Experience Design* al fine di poter realizzare una libreria che non fosse solamente utile allo sviluppatore, ma che fosse anche semplice ed intuitiva da utilizzare, fattore che abbiamo ritenuto molto importante. La fase preliminare infatti è stata un'analisi del problema iniziale specifico fornito dall'azienda, problema su cui abbiamo ragionato al fine di ottenere un'astrazione che ci permettesse di individuare un problema più grande a monte di tutto. Una volta individuato questo problema da risolvere ci siamo concentrati su una fase iniziale di *benchmark* in cui abbiamo cercato ed analizzato le soluzioni e le tecnologie esistenti utilizzabili per risolverlo, cercando

di scovare i punti di forza e le debolezze di ogni tecnologia. Terminate le fasi iniziali di analisi abbiamo iniziato la progettazione della soluzione e delle funzionalità di cui avevamo bisogno prima di passare al vero e proprio sviluppo di questa possibile soluzione, periodo durante il quale abbiamo alternato anche fasi di prototipazione e test per poter individuare malfunzionamenti, funzionalità mancanti oppure cose poco chiare ad un potenziale utilizzatore finale. Infine, l'ultimo step è stato lo sviluppo di un prototipo finale con cui abbiamo testato la versione definitiva della soluzione sviluppata e proposta. Grazie a questo prototipo, chiamato *WAY*, abbiamo dimostrato non solo quali sono le potenzialità della libreria *AR POI Experiences* che abbiamo sviluppato, ma anche quanto sia semplice utilizzarne le funzionalità: lo sviluppatore infatti non ha bisogno di avere esperienza nel campo della *Computer Graphics* per poter creare e configurare una intera scena in Realtà Aumentata all'interno della propria applicazione.

La soluzione sviluppata può essere sicuramente migliorata sotto molti aspetti, ad esempio si potrebbero implementare tecniche migliori per definire la similarità tra *Cluster* quando vogliamo aggiornare la scena effettuando una transizione tra la vecchia posizione e quella nuova appena calcolata, oppure si potrebbero utilizzare funzionalità di terzi per ottenere indicazioni stradali nel caso in cui lo sviluppatore voglia lavorare con *POIs Dinamici*. Un'altra funzionalità interessante che ci riserviamo per sviluppi futuri potrebbe essere quella di integrare algoritmi di *Computer Vision* ed *Image Processing* al fine di posizionare in modo migliore e più stabile i *Renderable*, ancorandoli ad esempio a pareti di edifici o a punti di riferimento che si trovano in corrispondenza del *POI* a cui sono associati. Le funzionalità che possiamo aggiungere e le loro sfumature sono davvero tantissime. Altri progetti che abbiamo riservato per sviluppi futuri riguardano la possibilità di inviare notifiche all'utente quando si trova nelle vicinanze di un *POI*, sfruttare le informazioni sull'attività svolta dall'utente in un particolare momento per fornirgli o bloccargli funzionalità, per esempio se ci accorgiamo che sta guidando possiamo avvertirlo dei pericoli che corre nell'utilizzare un device alla

guida, oppure semplicemente si potrebbero aggiungere molte più animazioni da applicare ai *Renderable* oltre alla transizione tra due punti, come un effetto di sparizione quando il *POI* viene rimosso dalla scena. Anche le librerie *Device Position* e *Device Orientation* potrebbero essere migliorate sotto vari aspetti, per esempio implementando politiche per ottimizzare il consumo di batteria o per migliorare la riduzione del rumore dei sensori tramite, ad esempio, tecniche di *sensor fusion* con cui combinare informazioni da più risorse.

Alla luce di ciò, possiamo quindi terminare questo lavoro con la consapevolezza che, pur sapendo che la soluzione sviluppata non è perfetta e che può essere sicuramente migliorata sotto molti aspetti, ci riteniamo particolarmente soddisfatti sia delle conoscenze acquisite grazie a questa esperienza sia del prodotto realizzato e concludiamo con la speranza che tutto il lavoro svolto possa essere utile anche solo come punto di partenza per la nascita di future soluzioni ed applicazioni che sfruttino la Realtà Aumentata allo scopo di semplificare e migliorare la vita delle persone.

Appendice A

Coordinate geografiche

Le coordinate geografiche[1] sono valori molto utili che ci permettono di individuare ed identificare un particolare punto sulla superficie terrestre tramite tre valori:

1. **latitudine**, distanza angolare[2] tra un punto e l'equatore;
2. **longitudine**, distanza angolare tra un punto ed il meridiano di *Greenwich*;
3. **altitudine**, distanza verticale tra un punto ed il livello del mare.

A differenza dell'altitudine che si esprime sempre in metri, la latitudine e la longitudine, dato che sono calcolate come distanze angolari, si esprimono in gradi. Quando indichiamo queste informazioni le specifichiamo sempre nell'ordine "*latitudine*, *longitudine*", ma possiamo utilizzare diversi formati per esprimere i loro valori:

[1]Esistono diversi sistemi di riferimento geodetico-cartografici con cui poter esprimere le coordinate geografiche. Nel nostro caso, quello utilizzato è il sistema *WGS84 UTM*, basato su una configurazione geocentrica e solidale con il nostro pianeta in cui il centro dell'ellissoide coincide esattamente con il centro di massa della Terra.

[2]Ricordiamo che la *distanza angolare* è la distanza tra due punti di una superficie sferica e si misura utilizzando l'ampiezza dell'angolo non concavo formato dai due raggi della sfera aventi come estremi i due punti considerati.

- **DMS** (*Degrees Minutes Seconds*), in cui tutto viene espresso utilizzando una *base sessagesimale*, un sistema di numerazione di origine babilonese utilizzato per la misurazione di tempo ed angoli; utilizzando questo formato esprimeremo, ad esempio, le coordinate del Ponte dei Sospiri di Venezia in questo modo: $N\,45°\,27'\,55.634"$, $E\,9°\,11'\,11.457"$; l'utilizzo delle lettere cardinali N ed E serve per indicare in quale emisfero si trova la coordinata (nel caso della latitudine) e per indicare se ci troviamo a sinistra o a destra del meridiano di *Greenwich* (nel caso della longitudine);

- **DM** (*Degrees Minutes*), formato molto meno comune dove non utilizziamo i secondi ma esprimiamo i minuti utilizzandone anche i decimali; in questo caso le coordinate del Ponte dei Sospiri di Venezia diventerebbero $N\,45°\,27.92724'$, $E\,9°\,11.190954'$;

- **DD** (*Decimal Degrees*), in cui esprimiamo le coordinate utilizzando i *gradi decimali* ed indichiamo con segno positivo latitudini nell'emisfero Nord e longitudini ad Est del meridiano di *Greenwich*, mentre con segno negativo indichiamo latitudini nell'emisfero Sud e longitudini ad Ovest del meridiano di *Greenwich*; con questo formato le coordinate del Ponte dei Sospiri di Venezia diventano: 45.465454, 9.1865159.

In Figura A.1[3] possiamo vedere una rappresentazione grafica di dove si trovano l'equatore, il meridiano di *Greenwich* ed i vari emisferi in cui è suddivisa la superficie terrestre.

Per comodità, per la creazione delle librerie *Device Orientation*, *Device Position* e *AR POI Experiences* abbiamo utilizzato il formato *DD*, ovvero la latitudine e la longitudine sono sempre espresse in *gradi decimali*. In particolare, è interessante il fatto che all'interno della libreria *AR POI Experiences* abbiamo anche implementato alcuni metodi per calcolare la coordinata GPS media di un insieme di punti e, calcolandola in modo automatico, il risultato

[3]L'immagine del globo è stata creata da *Encyclopedia Britannica*, `https://www.britannica.com`, ed è stata successivamente modificata per adattarla alle nostre esigenze.

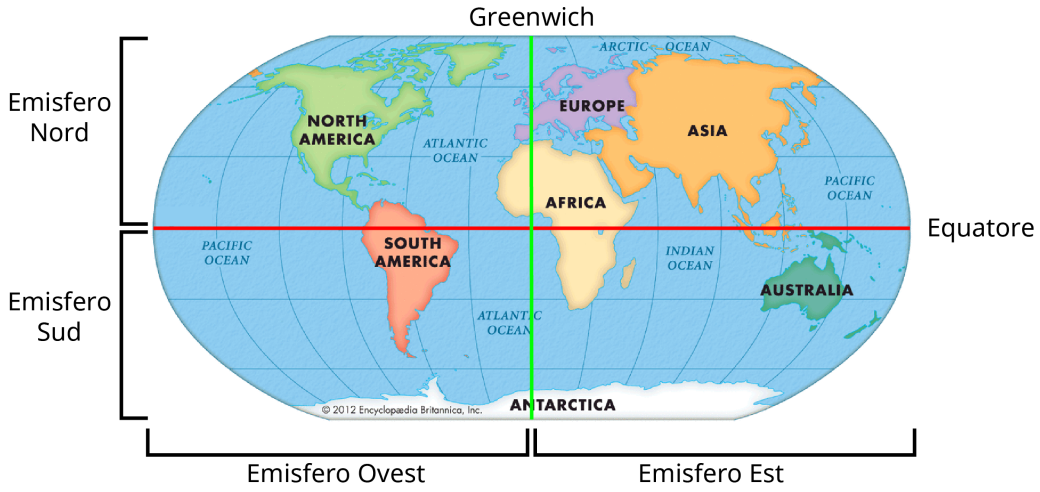

Figura A.1: Indicazione degli emisferi, del meridiano di *Greenwich* e dell'equatore.

ottenuto può avere un numero arbitrario di decimali, per cui ci siamo chiesti il significato di questi decimali. A seguito di uno studio approfondito abbiamo scoperto che ogni decimale ha un diverso significato: un grado corrisponde ad una precisione di circa 111 chilometri attorno all'equatore, per cui maggiore è il numero di decimali e maggiore è la precisione della coordinata GPS.

Utilizziamo le seguenti lettere come guida per spiegare il significato di ogni cifra che compone la latitudine o la longitudine e per fornire un esempio migliore del loro valore:

$$\pm ABC.DEFGHILMN...$$

- come detto precedentemente, il segno ci indica in quale emisfero ci troviamo (Figura A.1);
- la prima cifra, indicata con **A**, è utilizzata solamente dalla longitudine poiché ha un range di valori compreso tra $[-180.0, +180.0]$, mentre la latitudine è limitata nel range $[-90.0, +90.0]$;
- **B** corrisponde ad una precisione di circa 1110 km e ci indica in quale oceano o continente ci troviamo;

- **C** ha una precisione di circa 111 km ed è in grado di distinguere la posizione tra diversi stati o nazioni;
- utilizzando il primo decimale, indicato con **D**, otteniamo una posizione con una precisione pari a circa 11 km, il che ci permette di distinguere tra grandi città vicine;
- il secondo decimale, indicato con **E**, offre una precisione di circa 1.1 km, distinguendo quindi tra diverse piccole città vicine;
- **F** invece ha una precisione di 110 metri circa, tale da poter distinguere campi agricoli diversi;
- il decimale indicato dalla lettera **G** è la precisione utilizzata solitamente dai GPS commerciali, ovvero circa 11 metri;
- utilizzando anche **H** arriviamo ad una posizione precisa circa fino ad 1.1 metri, il che ci permette di distinguere anche due alberi differenti;
- la lettera **I** è il decimale che offre una precisione di circa 0.11 metri, adatta per tracciare ad esempio dei movimenti di ghiacciai oppure di argini di un fiume;
- il decimale indicato dalla lettera **L** è la precisione massima raggiunta dalla maggior parte delle tecniche basate su GPS, ovvero una precisione pari a circa 11 millimetri;
- utilizzando anche il decimale indicato dalla lettera **M** possiamo avere una precisione pari a circa 1.1 millimetri, tale per cui possiamo monitorare movimenti vulcanici oppure quelli delle placche tettoniche;
- infine continuando ad aggiungere decimali dalla lettera **N** in poi otteniamo delle precisioni microscopiche, per esempio il quindicesimo decimale avrebbe l'ordine di grandezza di un atomo.

Considerando questi fattori, per i metodi all'interno delle librerie sviluppate abbiamo troncato le coordinate GPS fino al settimo decimale compreso, ovvero fino ottenere una precisione di circa 11 millimetri.

Appendice B

Creazione di contenuti virtuali

Utilizzando *ARCore* e *Sceneform* disponiamo di diversi metodi utilizzabili per la creazione di un *Renderable*, ovvero di un modello 3D composto da vertici, facce, materiali, textures e molto altro che può essere renderizzato all'interno dell'ambiente in Realtà Aumentata. Di seguito, facendo riferimento alle nostre esperienze personali ed alla documentazione ufficiale[1], spiegheremo quali solo le tecniche consigliate non solo per la creazione di *Renderable*, ma anche per la creazione di *ViewRenderable* da utilizzare all'interno di una scena in AR.

B.1 Importazione di un modello 3D

Il modo più semplice e veloce per creare un contenuto virtuale 3D è quello di importarne l'*asset*[2] all'interno del progetto ed utilizzare i tools messi a disposizione da *Sceneform* per convertirlo nel formato *SFB* (Sceneform

[1]Sceneform APIs: Create Renderables, `https://developers.google.com/ar/develop/java/sceneform/create-renderables`.

[2]Nel mondo della *Computer Graphics*, con il termine *asset* ci riferiamo a tutto ciò che non è puro codice da eseguire, come modelli 3D, animazioni, texture, suoni o video. Nel nostro particolare caso, quando utilizziamo il termine *asset* ci riferiamo alla descrizione del modello utilizzato per creare il contenuto virtuale.

Binary asset), in modo tale da poterlo successivamente convertire in un *ModelRenderable*[3] all'interno del codice.

Per importare l'*asset* nel nostro progetto basta semplicemente creare una cartella chiamata *sampledata* all'interno della cartella *app* del nostro progetto. Come possiamo vedere in Figura B.1, in *Android Studio* lo possiamo fare dal menù *File* cliccando prima su *New* e poi su *Sample Data Directory*. Una

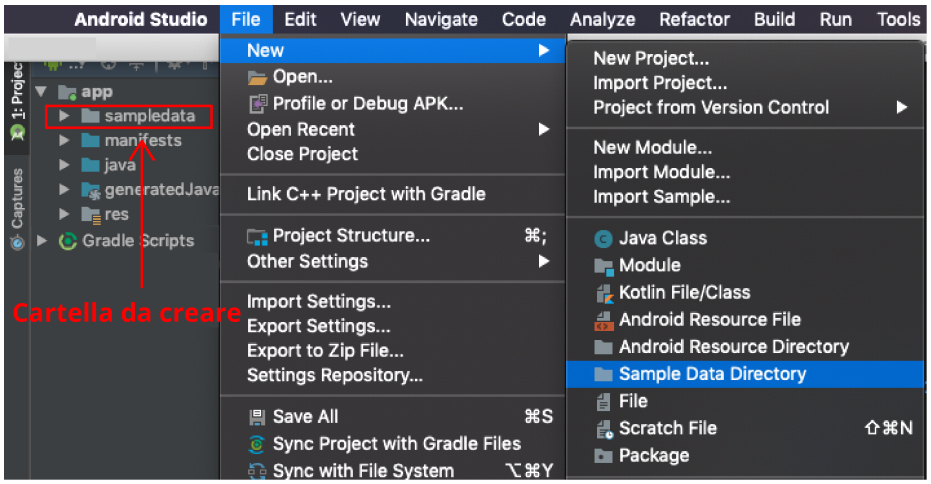

Figura B.1: Creazione della cartella *sampledata* su *Android Studio*. In blu vediamo evidenziata l'opzione sul menù da cliccare, mentre in rosso abbiamo indicato la cartella che si creerà come risultato.

volta creata possiamo copiare il nostro *asset* all'interno di questa cartella, questo perché quando andremo a creare l'*APK* della nostra applicazione, tutto il contenuto della cartella *sampledata* non verrà incluso, il che la rende il punto ideale in cui inserire questi modelli. Infine, l'ultimo passaggio necessario è quello di convertirli in formato *SFB* aggiungendo le seguenti righe alla fine del file *build.gradle*:

```
1 dependencies {
2     ...
3 }
```

[3]Sceneform APIs: ModelRenderable, `https://developers.google.com/ar/reference/java/com/google/ar/sceneform/rendering/ModelRenderable`.

```
apply plugin: 'com.google.ar.sceneform.plugin'
sceneform.asset('sampledata/modelname.obj',
  'default',
  'sampledata/modelname.sfa',
  'src/main/res/raw/modelname')
```

Con queste poche righe inserite all'interno del file *build.gradle* della nostra app abbiamo specificato che vogliamo utilizzare il plugin di *Sceneform* e che vogliamo convertire il nostro *asset*, chiamato *modelname.obj*, in un formato utilizzabile per la creazione del *Renderable*. Il risultato sarà memorizzato all'interno della cartella *raw* del nostro progetto.

B.2 Creazione del Renderable

Dopo aver seguito tutti i passaggi per importare correttamente un *asset* all'interno del nostro progetto possiamo occuparci creazione del *ModelRenderable* da poter associare ad uno dei *Node* da utilizzare all'interno della scena in AR. Per farlo sfruttiamo il *builder* associato alla classe *ModelRenderable*:

```
ModelRenderable.builder()
  .setSource(this, R.raw.modelname)
  .build()
  .thenAccept(renderable -> mModelRenderable = renderable)
  .exceptionally(throwable -> { /* Gestire eccezione. */ });
```

Come possiamo notare da queste righe di codice, dato che quando abbiamo convertito il nostro *asset* in formato *SFB* abbiamo specificato la cartella *raw* come destinazione dell'output, possiamo ricavare il modello direttamente da quella cartella specificando *R.raw.modelname*[4].

[4]In Android possiamo accedere a tutte le risorse costanti all'interno del nostro progetto sfruttando la classe *R*. Maggiori informazioni sulla documentazione ufficiale `https://developer.android.com/guide/topics/resources/providing-resources`.

La cosa interessante da evidenziare è l'oggetto di tipo *CompletableFuture*[5] restituito dal metodo *build()*: questo oggetto implementa l'interfaccia *Future*[6], la quale rappresenta il risultato di un calcolo asincrono. In particolare, il *CompletableFuture* è un *Future* che può essere esplicitamente completato, permettendoci quindi di definire diverse azioni o comportamenti da eseguire al termine dell'operazione asincrona.

Per nostra esperienza personale, quello che consigliamo di fare quando si crea un'applicazione in Realtà Aumentata è quello di definire una struttura dati che contenga tutti i risultati di tipo *CompletableFuture* di ogni metodo *build()* eseguito sui *ModelRenderable* di ogni contenuto che vogliamo renderizzare, in modo tale da poter sfruttare il metodo *allOf()*[7] per assicurarci che tutti i *ModelRenderable* vengano creati correttamente prima di utilizzarli all'interno della nostra scena:

```
1  /* Struttura dati che conterra' i CompletableFuture. */
2  CompletableFuture[] cfs = new CompletableFuture[2];
3  /* Eseguiamo il metodo build() su tutti gli asset. */
4  cfs[0] = ModelRenderable.builder()
5    .setSource(this, R.raw.model1).build();
6  cfs[1] = ModelRenderable.builder()
7    .setSource(this, R.raw.model2).build();
8  /* Specifichiamo cosa fare al completamento di build(). */
9  CompletableFuture.allOf(cfs).handle((unused, throwable) -> {
10   if (throwable != null) { /* Gestire eccezione. */ }
11   /* Ottengo i ModelRenderable come risultato. */
12   try {
13     mModelRenderable1 = (ModelRenderable) cfs[0].get();
14     mModelRenderable2 = (ModelRenderable) cfs[1].get();
15   }
```

[5] Android APIs: CompletableFuture, `https://developer.android.com/reference/java/util/concurrent/CompletableFuture`.

[6] Android APIs: Future, `https://developer.android.com/reference/java/util/concurrent/Future`.

[7] Android APIs: CompletableFuture allOf, `https://developer.android.com/reference/java/util/concurrent/CompletableFuture.html#allOf(java.util.concurrent.CompletableFuture<?>...)`.

```
16    catch (InterruptedException | ExecutionException e) {
17      /* Gestire eccezione. */
18    }
19    return null;
20  });
```

Altra nota importante da fare è che ogni *ModelRenderable* che viene creato può essere associato a più di un *Node* all'interno della scena, ma ad una condizione: tutte le modifiche che vengono fatte al *ModelRenderable* saranno poi propagate a tutti i *Node* della scena a cui è applicato. Questo vuol dire che avranno tutti la stessa grandezza, lo stesso colore, lo stesso orientamento e così via. Nel caso in cui si voglia utilizzare lo stesso *asset* per più *Node* di una scena, ma vogliamo avere la certezza di poterli gestire in modo indipendente, dobbiamo ricordarci di creare tanti *ModelRenderable* quanti sono i *Node* a cui vogliamo applicarlo specificando lo stesso *asset* all'interno del metodo *setSource*[8] di tutti i *ModelRenderable*.

B.3 Creare forme semplici a runtime

Nella documentazione ufficiale[9] troviamo anche un esempio di come poter creare a runtime semplici forme 3D da poter utilizzare come contenuto virtuale, tecnica che non richiede la creazione e l'importazione di un apposito *asset* all'interno del nostro progetto. La creazione di queste forme avviene utilizzando due funzionalità fornite da *Sceneform*:

- la prima si chiama *ShapeFactory*[10] e ci permette di creare dinamicamente dei *ModelRenderable* utilizzando semplici forme come sfere o

[8]Sceneform APIs: ModelRenderable setSource, `https://developers.google.com/ar/reference/java/com/google/ar/sceneform/rendering/ModelRenderable.Builder#setSource(com.google.ar.sceneform.rendering.RenderableDefinition)`.

[9]Create simple shapes at runtime, `https://developers.google.com/ar/develop/java/sceneform/create-renderables#create_simple_shapes_at_runtime`.

[10]Sceneform APIs: ShapeFactory, `https://developers.google.com/ar/reference/java/com/google/ar/sceneform/rendering/ShapeFactory`.

cubi;

- la seconda invece si chiama *MaterialFactory*[11] e ci permette di creare e definire dei materiali da applicare sulle forme create.

Utilizzare queste due funzionalità è molto semplice, ad esempio per la creazione di una sfera di colore rosso ci bastano le seguenti righe di codice:

```
1 /* Definiamo un materiale opaco di colore rosso. */
2 MaterialFactory.makeOpaqueWithColor(this,
3     new Color(android.graphics.Color.RED)
4   ).thenAccept(material -> {
5     /* Creiamo un ModelRenderable di forma sferica. */
6     redSphereRenderable = ShapeFactory.makeSphere(
7       0.1f,   // centro della sfera
8       new Vector3(0.0f, 0.15f, 0.0f), // dimensioni
9       material  // materiale
10    );
11  });
```

B.4 Creare ViewRenderable

Un'altra delle funzionalità interessanti offerte da *Sceneform* è la possibilità di trasformare i *widget* standard di Android in contenuti virtuali 2D da utilizzare all'interno di una scena in AR. Questi contenuti, che abbiamo utilizzato anche all'interno della libreria *AR POI Experiences*, vengono chiamati *ViewRenderable* e li possiamo creare in modo molto simile a quello descritto precedentemente per i *ModelRenderable*.

Per prima cosa dobbiamo creare un *layout* che contenga il *widget* che vogliamo utilizzare all'interno della nostra scena in Realtà Aumentata. Ad esempio, per sviluppare il prototipo *WAY* descritto nel Capitolo 5, abbiamo definito un *layout* formato da due *TextView*, la prima utilizzata come titolo e la seconda come descrizione:

[11] Sceneform APIs: MaterialFactory, `https://developers.google.com/ar/reference/java/com/google/ar/sceneform/rendering/MaterialFactory`.

```
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:orientation="vertical">
  <TextView
    android:id="@+id/title"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="Title"/>
  <TextView
    android:id="@+id/description"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="Description" />
</LinearLayout>
```

Dopo aver definito il *layout* da utilizzare possiamo creare il *ViewRenderable* utilizzando l'apposito metodo *builder*:

```
ViewRenderable.builder()
  .setView(this, R.layout.mylayoutname)
  .build()
  .thenAccept(renderable -> mViewRenderable = renderable);
```

Anche in questo caso, consigliamo di adottare i due stratagemmi descritti precedentemente per la creazione di un *ModelRenderable*:

1. per la creazione sfruttare una struttura dati che contenga tutti i *CompletableFuture* ottenuti dai metodi *build()*, in modo tale da poter successivamente definire le azioni ed i comportamenti da utilizzare sia nel caso di completamento con successo che nel caso in cui l'operazione abbia generato delle eccezioni;

2. se vogliamo gestire i *ViewRenderable* in modo indipendente, ad esempio vogliamo che ognuno abbia contenuti differenti, dobbiamo ricordarci di

sfruttare lo stesso *layout* per creare tanti *ViewRenderable* quanti sono i contenuti virtuali di cui abbiamo bisogno.

Appendice C

Formula di Vincenty

Come descritto nel Paragrafo 4.6.1, la *Formula di Vincenty* è composta da due differenti approcci in base al risultato che si vuole ottenere:

1. *Metodo Diretto*, utilizzato per calcolare la coordinata GPS di un particolare punto data l'angolazione e la distanza da un'altra coordinata;

2. *Metodo Inverso*, utilizzato per calcolare la lunghezza della *geodetica* tra una coppia di coordinate GPS.

Ricordiamo che con il termine *geodetica* si intende la curva di minor lunghezza che congiunge due punti di uno spazio e giace sullo spazio stesso

Per gli scopi di questa tesi abbiamo implementato e testato il *Metodo Inverso* della *Formula di Vincenty* per poterlo confrontare con la *Formula dell'emisenoverso* e decidere quale delle due tecniche utilizzare. Prima di passare alla spiegazione dei passi utilizzati da questo metodo, è necessario definire la notazione utilizzata per l'algoritmo:

- a e b sono, rispettivamente, il semiasse maggiore e quello minore dell'ellissoide secondo il sistema di riferimento *WGS84*[1] *UTM*[2], utilizzato solitamente dai GPS;

[1]World Geodetic System: WGS84, `https://gisgeography.com/wgs84-world-geodetic-system/`.

[2]La sigla *UTM* viene utilizzata per indicare il sistema di coordinate *Universal Transverse Mercator*.

- $f = \dfrac{a-b}{a}$ è l'ellitticità dell'ellissoide utilizzato come modello della Terra;
- ϕX e μX indicano, rispettivamente, la latitudine e la longitudine di un generico punto X;
- A e B sono le due coordinate GPS di cui vogliamo calcolare la distanza;
- $L = \mu B - \mu A$ è la differenza di longitudine;
- σ e σ_m sono, rispettivamente, la distanza angolare tra i punti A e B e la distanza angolare tra l'equatore ed il punto medio della linea che collega i due punti;
- λ è la differenza in longitudine tra i due punti;
- U_1 e U_2 sono, rispettivamente, la latitudine ridotta del punto A e del punto B; preso un generico punto X definito da latitudine e longitudine, la latitudine ridotta è definita come $U = (1-f)\tan\phi$;
- α rappresenta l'angolo di azimut della *geodetica* all'equatore;
- s è la lunghezza della *geodetica*, ovvero il risultato che vogliamo trovare utilizzando questo algoritmo.

Lo step iterativo del metodo consiste nel calcolare ripetutamente il valore di λ e confrontarlo con il risultato ottenuto allo step precedente: quando la differenza tra i due λ sarà trascurabile, in base al grado di accuratezza che abbiamo definito, allora possiamo procedere al calcolo della *geodetica*. Al primo step, si suppone che λ sia pari L:

$$\sin\sigma = \sqrt{(\cos U_2 \sin\lambda)^2 + (\cos U_1 \sin U_2 - \sin U_1 \cos U_2 \cos\lambda)^2}; \tag{C.1}$$

$$\cos\sigma = \sin U_1 \sin U_2 + \cos U_1 \cos U_2 \cos\lambda; \tag{C.2}$$

$$\sigma = \arctan\frac{\sin(\sigma)}{\cos(\sigma)}; \tag{C.3}$$

$$\sin\alpha = \frac{\cos U_1 \cos U_2 \sin\lambda}{\sin\sigma}; \tag{C.4}$$

$$\cos(2\sigma_m) = \cos\sigma - \frac{2\sin U_1 \sin U_2}{\cos^2\alpha}; \tag{C.5}$$

$$C = \frac{f}{16}\cos^2\alpha[4 + f(4 - 3\cos^2\alpha)]; \tag{C.6}$$

$$\lambda = L + (1 - C)f\sin\alpha\{\sigma + C\sin\alpha[\cos(2\sigma_m) + C\cos\sigma(-1 + 2\cos^2(2\sigma_m))]\}; \tag{C.7}$$

Al termine del processo iterativo possiamo utilizzare i valori appena calcolati al suo interno per ricavare la lunghezza s della *geodetica*:

$$u^2 = \cos^2\alpha\frac{a^2 - b^2}{b^2}; \tag{C.8}$$

$$A = 1 + \frac{u^2}{16384}\{4096 + u^2[-768 + u^2(320 - 175u^2)]\}; \tag{C.9}$$

$$B = \frac{u^2}{1024}\{256 + u^2[-128 + u^2(74 - 47u^2)]\}; \tag{C.10}$$

$$\Delta\sigma = B\sin\sigma\{\cos(2\sigma_m) + \frac{1}{4}B[\cos\sigma(-1 + 2\cos^2(2\sigma_m)) - \frac{1}{6}B\cos(2\sigma_m)(-3 + 4\sin^2\sigma)(-3 + 4\cos^2(2\sigma_m))]\}; \tag{C.11}$$

$$s = bA(\sigma - \Delta\sigma); \tag{C.12}$$

Bibliografia

[1] Statista The Statistics Portal. Global digital population as of October 2018 (in millions). `https://www.statista.com/statistics/617136/digital-population-worldwide/`. Online, consultata il 03 Dicembre 2018.

[2] The Global Leader in Games and Newzoo Esports Analytics. Newzoo's 2018 Global Mobile Market Report: Insights into the World's 3 Billion Smartphone Users. `https://newzoo.com/insights/articles/newzoos-2018-global-mobile-market-report-insights-into-the-worlds-3-billion-smartphone-users/`. Online, consultata il 03 Dicembre 2018.

[3] Vito Di Bari and Paolo Magrassi. *2015 weekend nel futuro. Viaggio nelle tecnologie che stanno per cambiare la nostra vita.* Il Sole 24 Ore, December 2004.

[4] Ronald Azuma. A Survey of Augmented Reality. *Presence: Teleoper. Virtual Environ.*, 6(4):355–385, August 1997.

[5] Ronald Azuma, Yohan Baillot, Reinhold Behringer, Steven Feiner, Simon Julier, and Blair MacIntyre. Recent Advances in Augmented Reality. *IEEE Comput. Graph. Appl.*, 21(6):34–47, November 2001.

[6] Julie Carmigniani, Borko Furht, Marco Anisetti, Paolo Ceravolo, Ernesto Damiani, and Misa Ivkovic. Augmented Reality Technologies, Sy-

stems and Applications. *Multimedia Tools Appl.*, 51(1):341–377, January 2011.

[7] Paul Milgram, Haruo Takemura, Akira Utsumi, and Fumio Kishino. Augmented Reality: A Class of Displays on the Reality-Virtuality Continuum. pages 282–292, 1994.

[8] Mobile App Daily. What's The Difference Between AR, MR and VR? `https://www.mobileappdaily.com/2018/09/13/difference-between-ar-mr-and-vr`. Online, consultata il 12 Novembre 2018.

[9] J. P. Gownder, Christopher Voce, Michelle Mai, and Diane Lynch. Breakout Vendors: Virtual And Augmented Reality, May 2016.

[10] L. Frank Baum. *The Master Key.* Bowen-Merril Company, 1901.

[11] Morton Heiling. Inventor in the field of Virtual Reality, Sensorama Machine. `http://www.mortonheilig.com/InventorVR.html`. Online, consultata il 12 Novembre 2018.

[12] Ivan E. Sutherland. A Head-mounted Three Dimensional Display. In *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*, AFIPS '68 (Fall, part I), pages 757–764, New York, NY, USA, 1968. ACM.

[13] Giuliano Ambrosio. La Realtà Aumentata cambierà il mondo e le nostre abitudini. `https://medium.com/@aquest/la-realt%C3%A0-aumentata-cambier%C3%A0-il-mondo-e-le-nostre-abitudini-7d9d678477df`. Online, consultata il 12 Novembre 2018.

[14] Myron Krueger. Videoplace (1985). `https://www.youtube.com/watch?time_continue=7&v=d4DUIeXSEpk`. Online, consultata il 12 Novembre 2018.

[15] Gavan Lintern. Transfer of Landing Skill after Training with Supplementary Visual Cues. *Human Factors*, 22(1):81–88, 1980. PMID: 7364448.

[16] David Daw. A Brief History of Wearable Computers. `https://www.pcworld.com/article/237241/wearable_tech.html`. Online, consultata il 12 Novembre 2018.

[17] Layar. How to Make Money with Augmented Reality in Publishing. `https://www.slideshare.net/layarmobile/layar-webinar-november-14`. Online, consultata il 12 Novembre 2018.

[18] L. Robert George, Douglas B.; Morris. A Computer-Driven Astronomical Telescope Guidance and Control System with Superimposed Star Field and Celestial Coordinate Graphics Display. *Journal of the Royal Astronomical Society of Canada (ISSN 0035-872X)*, 83:32–41, February 1989.

[19] Caudell. Thomas and David W. Mizell. Augmented Reality: An Application of Heads-Up Display Technology to Manual Manufacturing Processes. 1992.

[20] Louis B. Rosenberg. Virtual Fixtures: Perceptual tools for telerobotic manipulation. *1993 IEEE Annual Virtual Reality International Symposium*, pages 76 – 82, 10 1993.

[21] Steven Feiner, Blair Macintyre, and Doree Seligmann. Knowledge-based Augmented Reality. *Commun. ACM*, 36(7):53–62, July 1993.

[22] Houchard J. Puccetti M. Abernathy, M. and J. Lambert. Debris Correlation Using the Rockwell WorldView System. In *Proceedings of 1993 Space Surveillance Workshop*, pages 189–195, April 1993.

[23] Jun Rekimoto and Yuji Ayatsuka. CyberCode: Designing Augmented Reality Environments with Visual Tags. In *Proceedings of DARE 2000*

on Designing Augmented Reality Environments, DARE '00, pages 1–10, New York, NY, USA, 2000. ACM.

[24] Hideyuki Tamura. Overview and Final Results of the MR Project. *Mixed Reality System Laboratory Inc.*

[25] Columbia University. The Touring Machine, Prototyping 3D mobile AR Systems for Exploring the Urban Environment. `https://blairmacintyre.me/project/touring-machine/`. Online, consultato il 25 Ottobre 2018.

[26] S. Feiner, B. MacIntyre, T. Hollerer, and A. Webster. A Touring Machine: Prototyping 3D Mobile Augmented Reality Systems for Exploring the Urban Environment. *Digest of Papers. First International Symposium on Wearable Computers*, October 1997.

[27] Ramesh Raskar, Greg Welch, and Henry Fuchs. Spatially Augmented Reality. In *Proceedings of the International Workshop on Augmented Reality: Placing Artificial Objects in Real Scenes*, IWAR '98, pages 63–72, Natick, MA, USA, 1999. A. K. Peters, Ltd.

[28] U.S. Naval Research Laboratory. Battlefield Augmented Reality System (BARS). `https://www.nrl.navy.mil/itd/imda/research/5581/augmented-reality`. Online, consultato il 25 Ottobre 2018.

[29] Frank J. Delgado, Michael F. Abernathy, Janis White, and William H. Lowrey. Real-time 3D Flight Guidance with Terrain for the X-38. volume 3691, pages 149–156, 1999.

[30] Frank J. Delgado, Scott Altman, Michael F. Abernathy, and Janis White. Virtual Cockpit Window for the X-38 Crew Return Vehicle. volume 4023, pages 63–70, June 2000.

[31] Steve Mann, James Fung, and Eric Moncrieff. EyeTap Technology for Wireless Electronic News Gathering. *SIGMOBILE Mob. Comput. Commun. Rev.*, 3(4):19–26, October 1999.

[32] Wearable Computer Lab. ARQuake: Interactive Outdoor Augmented Reality Collaboration System. `https://wearables.unisa.edu.au/projects/arquake/`. Online, consultata il 12 Novembre 2018.

[33] G. Reitmayr and D. Schmalstieg. Mobile Collaborative Augmented Reality. In *Proceedings of the IEEE and ACM International Symposium on Augmented Reality (ISAR'01)*, ISAR '01, pages 114–, Washington, DC, USA, 2001. IEEE Computer Society.

[34] Vassilios Vlahakis, John Karigiannis, Manolis Tsotros, Michael Gounaris, Luis Almeida, Didier Stricker, Tim Gleue, Ioannis T. Christou, Renzo Carlucci, and Nikos Ioannidis. Archeoguide: First Results of an Augmented Reality, Mobile Computing System in Cultural Heritage Sites. In *Proceedings of the 2001 Conference on Virtual Reality, Archeology, and Cultural Heritage*, VAST '01, pages 131–140, New York, NY, USA, 2001. ACM.

[35] Trimble Navigation and HITLab. Sistema outdoor con AR montato su casco. `https://www.youtube.com/watch?v=jL3C-OVQKWU`. Online, consultato il 29 Ottobre 2018.

[36] BMW Group. The new MINI Cabrio, Launch Campaign with Augmented Reality Technology. `https://www.press.bmwgroup.com/global/photo/detail/P0051421/the-new-mini-cabrio-launch-campaign-with-augmented-reality-technology-11-2008`. Online, consultato il 12 Novembre 2018.

[37] Paul Strauss. Mini Augmented Reality Ads Hit Newstands. `https://technabob.com/blog/2008/12/17/mini-augmented-reality-ads-hit-newstands/`. Online, consultata il 12 Novembre 2018.

[38] Mashable. Esquire Brings Augmented Reality to Print. `https://mashable.com/2009/10/30/esquire-augmented-reality/?europe=true#vtkJesK4TmqZ`. Online, consultato il 24 Ottobre 2018.

[39] Meta. Meta 1 Developer Kit — Kickstarter. https://www.kickstarter.com/projects/551975293/meta-the-most-advanced-augmented-reality-interface. Online, consultato il 25 Ottobre 2018.

[40] Volkswagen. MARTA: Mobile Augmented Reality Technical Assistance. https://www.volkswagenag.com/en/group/research/virtual-technologies.html#. Online, consultato il 25 Ottobre 2018.

[41] iDB. New Audi App uses Augmented Reality to help you find stuff on your car. https://www.idownloadblog.com/2013/08/13/audi-iphone-app-augmented/. Online, consultato il 29 Ottobre 2018.

[42] Android World. Google Glass Explorer Edition. https://www.androidworld.it/recensioni/google-glass/. Online, consultata il 12 Novembre 2018.

[43] BlippAR. BlippAR Creates First True AR Games for Google Glass and Announces Launch of Gaming Platform. https://www.blippar.com/blog/2014/05/29/blippar-creates-first-true-ar-games-for-google-glass-announces-launch-of-gaming-platform-. Online, consultato il 29 Ottobre 2018.

[44] The Verge. Google announces Project Tango, a smartphone that can map the world. https://www.theverge.com/2014/2/20/5430784/project-tango-google-prototype-smartphone-announced. Online, consultato il 29 Ottobre 2018.

[45] The Verge. Google leads $542 million funding of mysterious augmented reality firm Magic Leap. https://www.theverge.com/2014/10/21/7026889/magic-leap-google-leads-542-million-investment-in-augmented-reality-startup. Online, consultato il 29 Ottobre 2018.

[46] Dealbook. Google Invests Heavily in Magic Leap's Effort to Blend Illusion and Reality. https://dealbook.nytimes.com/2014/10/21/

google-invests-in-magic-leap-an-augmented-reality-firm/. Online, consultato il 25 Ottobre 2018.

[47] Augment. Infographic: The History of Augmented Reality. http://www.augment.com/blog/infographic-lengthy-history-augmented-reality/. Online, consultato il 23 Ottobre 2018.

[48] GPS.gov Official U.S. government information about the Global Positioning System (GPS) and related topics. GPS Accuracy. https://www.gps.gov/systems/gps/performance/accuracy/. Online, consultata il 25 Ottobre 2018.

[49] W3C. Main Page - Point of Interest Working Group. https://www.w3.org/2010/POI/wiki/Main_Page. Online, consultata il 24 Ottobre 2018.

[50] David Douglas, Demetri Venets, Cliff Wilke, David Gibson, Lance Liotta, Emanuel Petricoin, Buddy Beck, and Robert Douglas. Augmented Reality and Virtual Reality: Initial Successes in Diagnostic Radiology. 05 2018.

[51] W. Li, A.Y.C. Nee, and S.K. Ong. A State-of-the-Art Review of Augmented Reality in Engineering Analysis and Simulation. *Multimodal Technologies Interact*, 2017.

[52] Alexandra Klimova, Anna Bilyatdinova, and Andrey Karsakov. Existing Teaching Practices in Augmented Reality. In *7th International Young Scientist Conference on Computational Science*. Procedia Computer Science, March 2018.

[53] Nicolas F. Gazcon, Juan M. Trippel Nagel, Ernesto A. Bjerg, and Silvia M. Castro. Fieldwork in Geosciences assisted by ARGeo: A mobile Augmented Reality system. *Computers and Geosciences*, 121:30–38, 2018.

[54] Statista The Statistics Portal. Forecast augmented (AR) and virtual reality (VR) market size worldwide from 2016 to 2022 (in billion U.S.

dollars). https://www.statista.com/statistics/591181/global-augmented-virtual-reality-market-size/. Online, consultata il 29 Ottobre 2018.

[55] Engadget. For Nike, augmented reality is the perfect way to sell hyped sneakers. https://www.engadget.com/2017/11/06/nike-snkrs-augmented-reality-bots/. Online, consultata il 29 Ottobre 2018.

[56] Evan Selleck. Instagram Rolls Out AR Face Filters. http://www.iphonehacks.com/2017/05/instagram-rolls-ar-face-filters.html. Online, consultata il 12 Novembre 2018.

[57] Ryan Christoffel. Snapchat Introduces New AR Features: World Lenses. https://www.macstories.net/news/snapchat-introduces-new-ar-feature-world-lenses/. Online, consultata il 12 Novembre 2018.

[58] Exposure. Nike's experiment with Facebook Messenger AR. https://exposurehq.com.au/blog/2018/05/18/nikes-experiment-with-facebook-messenger-ar/. Online, consultata il 12 Novembre 2018.

[59] Rüdiger Pryss, Philip Geiger, Marc Schickler, Johannes Schobel, and Manfred Reichert. Advanced Algorithms for Location-Based Smart Mobile Augmented Reality Applications. *Procedia Computer Science*, 94:97–104, 2016. The 11th International Conference on Future Networks and Communications (FNC 2016) / The 13th International Conference on Mobile Systems and Pervasive Computing (MobiSPC 2016) / Affiliated Workshops.

[60] Rüdiger Pryss, Marc Schickler, Johannes Schobel, Micha Weilbach, Philip Geiger, and Manfred Reichert. Enabling Tracks in Location-Based Smart Mobile Augmented Reality Applications. *Procedia Computer Science*, 110:207–214, 2017. 14th International Conference on Mobile

Systems and Pervasive Computing (MobiSPC 2017) / 12th International Conference on Future Networks and Communications (FNC 2017) / Affiliated Workshops.

[61] Qualcomm. Qualcomm Augmented Reality. `https://www.qualcomm.com/invention/cognitive-technologies/immersive-experiences/augmented-reality`. Online, consultata il 29 Ottobre 2018.

[62] ScopeAR, Putting Knowledge Where You Need It. `https://www.scopear.com/`. Online, consultata il 29 Ottobre 2018.

[63] Niantic Inc. A Peek Inside the Niantic Real World Platform. `https://www.nianticlabs.com/blog/nianticrealworldplatform/`. Online, consultata il 29 Ottobre 2018.

[64] BlippAR. Urban Visual Positioning - Our Breakthrough in Location-based AR. `https://www.blippar.com/blog/2017/08/02/introducing-urban-visual-positioning-breakthrough-location-based-ar`. Online, consultato il 29 Ottobre 2018.

[65] Wikitude. Wikitude Pricing. `https://www.wikitude.com/store/`. Online, consultata il 12 Novembre 2018.

[66] Google. ARCore Fundamental Concepts. `https://developers.google.com/ar/discover/concepts`. Online, consultata il 12 Novembre 2018.

[67] Google. ARCore: Create Renderables. `https://developers.google.com/ar/develop/java/sceneform/create-renderables`. Online, consultata il 13 Novembre 2018.

[68] Google. Just a Line. `https://justaline.withgoogle.com`. Online, consultata il 13 Novembre 2018.

[69] Android. Distribution Dashboard. `https://developer.android.com/about/dashboards/`. Online, consultata il 26 Ottobre 2018.

[70] T. Vincenty. Direct and Invers Solutions of Geodesics on the Ellipsoid with Application of Nested Equations. *Survey Review*, 23(176):88–93, 1975.

Ringraziamenti

Vorrei spendere due parole di ringraziamento nei confronti di tutte le persone che mi hanno aiutato e sostenuto durante il mio percorso di studio e di tesi. Prima di tutto, vorrei ringraziare il Prof. Luciano Bononi ed il Dott. Luca Bedogni per tutto l'aiuto ed i preziosi consigli che mi hanno fornito in questo periodo e per la loro continua disponibilità.

Un ringraziamento particolare va a tutti i ragazzi di Comuni-Chiamo che mi hanno fornito tutti gli strumenti di cui avevo bisogno per portare a compimento questo lavoro. In particolare vorrei ringraziare Matteo, Andrea e Jacopo per i consigli tecnici e le importanti riflessioni sulle funzionalità, Danila per la pazienza e la disponibilità nel capire e risolvere i miei litigi con Illustrator e Gilberto per avermi fornito un ottimo Mac su cui poter lavorare alla tesi. Grazie a tutti voi per la vostra ospitalità e per questa bellissima esperienza in vostra compagnia.

Vorrei ringraziare Daniele, Deiv, Stefano e Marc per questi anni passati assieme in Università, per avermi fatto scoprire il buonissimo sushi di San Itacho e per tutte le partite a Miseria al settimo piano, ma soprattuto per tutti i bei ricordi che abbiamo creato assieme. Ringrazio di cuore tutti i miei amici, anche se siete troppi per citarvi tutti sappiate che sono davvero onorato di avervi conosciuto e delle belle serate assieme. Un ringraziamento speciale vorrei farlo ai miei amici più cari: a Diego per le sue idee strampalate senza nessun senso logico, a VAlessio per le chiacchierate, il fantacalcio e le giornate in compagnia ed a Luca per tutte le riflessioni, l'aiuto, il supporto ed i consigli che mi ha dato nel corso di questo periodo. Ci conosciamo da

tantissimo tempo, ma nonostante i nostri alti e bassi non smetterò mai di volervi bene.

Un grande ringraziamento a Franca, Enrico e Luciano perché senza il loro aiuto ed il loro sostegno non sarei mai arrivato fin qui, non vi ringrazierò mai abbastanza per l'opportunità che mi avete dato e per tutto quello che avete fatto per me. Un ringraziamento di cuore vorrei farlo anche ai miei genitori, che hanno creduto in me fino ad oggi e mi hanno dato la possibilità di inseguire i miei sogni, e a mio fratello Nicola, a cui voglio un bene indescrivibile. Sono orgoglioso dell'uomo che stai diventando. Grazie di cuore a tutti quanti.

Infine vorrei ringraziare l'unica persona in grado di capirmi veramente, di sapermi ascoltare e di sopportarmi sempre. In tutti questi anni, sia nei momenti di sconforto che in quelli felici, quando avevo bisogno di una mano eri sempre al mio fianco, pronta ad aiutarmi offrendomi un nuovo punto di vista e saggi consigli, ma soprattutto mi hai sempre dato la forza e l'energia per continuare e per non arrendermi nei periodi più faticosi. Insieme abbiamo pianto e riso, abbiamo guardato un'infinità di serie tv e film ed abbiamo visitato posti stupendi. Senza di te non credo proprio che sarei riuscito ad arrivare fin qui e farò il possibile per aiutarti a realizzare i tuoi sogni. Mi hai reso la persona che sono oggi, grazie di tutto amore mio.

Un grazie di cuore a tutti quanti,
Davide Marchi.