

AR Dice

*Un'applicazione mobile multi-piattaforma per lancio
di dadi in realtà aumentata*

GitHub Link

<https://github.com/Federicoand98/AR-Dice>

Andrucci Federico, Gianelli Alex, Righi Lorenzo
12/12/2021

Introduzione

Abstract

L’obiettivo del nostro progetto consiste nella realizzazione di un ambiente in realtà aumentata con il quale interagire con dadi virtuali. Per rendere questo connubio il più naturale e realistico possibile abbiamo sfruttato il *Game Engine Unity*, il quale ci permette di sfruttare componenti di AR inclusi nel package *AR Foundation* e le simulazioni fisiche dell’engine stesso. Inoltre, essendo sviluppato in Unity, può essere utilizzato per molteplici tipologie di dispositivi anche se il nostro focus è stato posto su *Android* e *iOS*.

Obiettivi

- Riconoscimento delle superfici e utilizzo delle stesse per simulare collisioni
- Individuazione e tracciamento di una lista predefinita di immagini, utilizzate per il posizionamento di oggetti virtuali
- Creazione e modifica di un tavolo da gioco personale
- Gestione della visualizzazione e della casualità dei risultati
- Modalità di lancio singola e multipla tramite l’utilizzo di dedicati *preset* modificabili dall’utente
- **Extra** Pagina di *Help* che illustra le modalità di utilizzo dell’applicazione e possibilità di modificare i colori del materiale dei dadi

Indice

1	Unity	1
1.1	Component-Based Model Design Pattern	2
1.1.1	GameObject e Components	2
2	AR Foundation	5
2.1	Architettura di AR Foundation	6
2.2	ARCore & ARKit	6
3	AR Dice	8
3.1	AR Foundation: componenti impiegati	9
3.1.1	ARSession	9
3.1.2	ARSessionOrigin & ARCamera	9
3.1.3	ARPlaneManager	10
3.1.4	ARRaycastManager	11
3.1.5	AROcclusionManager	11
3.1.6	ARTrackedImageManager	12
3.2	Struttura Software	13
3.2.1	Data Container	13
3.2.2	Game Controller	14
3.2.3	Table Controller	21
3.2.4	Image Tracking Controller	26
3.2.5	Settings Controller e Persistance Controller	28
4	Conclusioni e sviluppi futuri	32

Capitolo 1

Unity

Unity è un **game engine** e IDE multi piattaforma che permette di sviluppare videogiochi, applicativi ed esperienze in 2D, 2.5D o 3D. Come game engine Unity fornisce agli sviluppatori tutti i tool necessari allo sviluppo di videogiochi. Tra questi vi sono fisica, 3D rendering, collision detection, etc.

Il core di Unity è scritto nei linguaggi di programmazione C e C++ mentre, per quanto riguarda lo sviluppo di giochi attraverso questo game engine, vengono fornite delle librerie scritte in C#, le quali fungono da *wrapper* esterno avvolgendo il core del motore di gioco.

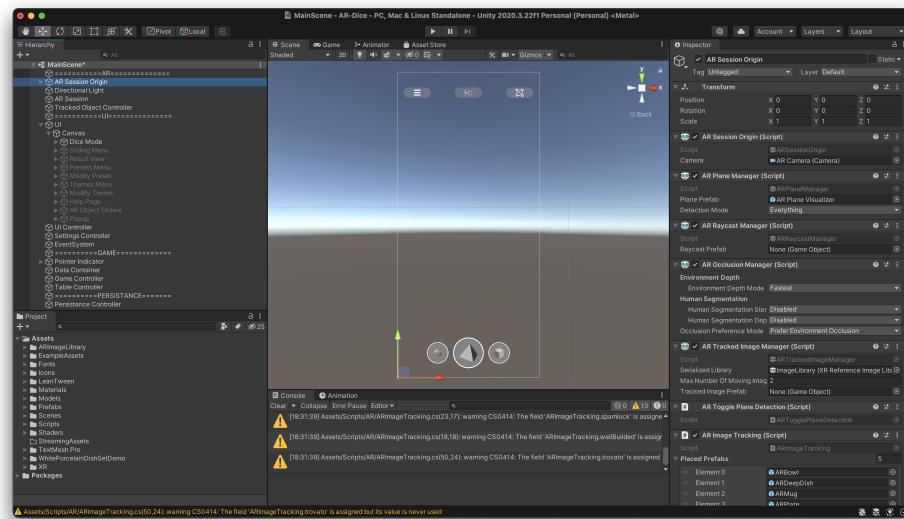


Figura 1.1: Editor di Unity

1.1 Component-Based Model Design Pattern

Il game engine di Unity è fondato sul design pattern **Component-Based Model** il quale va a cambiare radicalmente il modo di pensare e programmare un videogioco. Infatti viene superato il tradizionale modo di programmare object oriented attraverso l'introduzione di **Componenti** ed **Entità**.

I due elementi fondamentali di questo design pattern, e quindi dell'architettura di Unity, sono:

- **GameObject**: i quali rappresentano le entità, ovvero gli oggetti, presenti nella scena;
- **MonoBehaviour**: i quali rappresentano i componenti, cioè il comportamento che le entità devono assumere.

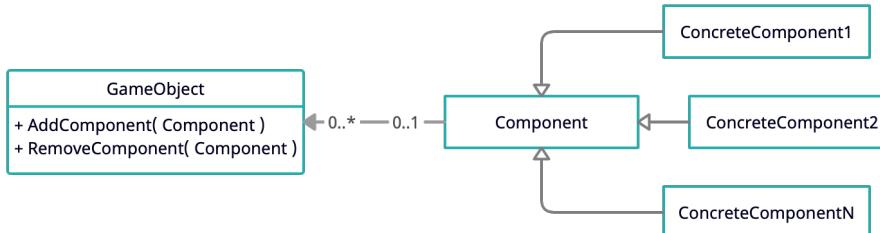


Figura 1.2: Component-Based Model design pattern

1.1.1 GameObject e Components

In Unity, attraverso il pattern Component-Based Model, i **GameObject** sono le entità principali per rappresentare qualsiasi oggetto, proprietà o scenario all'interno del mondo di gioco.

Anche se apparentemente sono elementi completamente scollegati tra di loro, fungono tutti da container per i **Component**, i quali implementano il comportamento e le funzionalità dello specifico GameObject.

Un esempio efficace di GameObject all'interno di **ARDice** può essere proprio un dado.

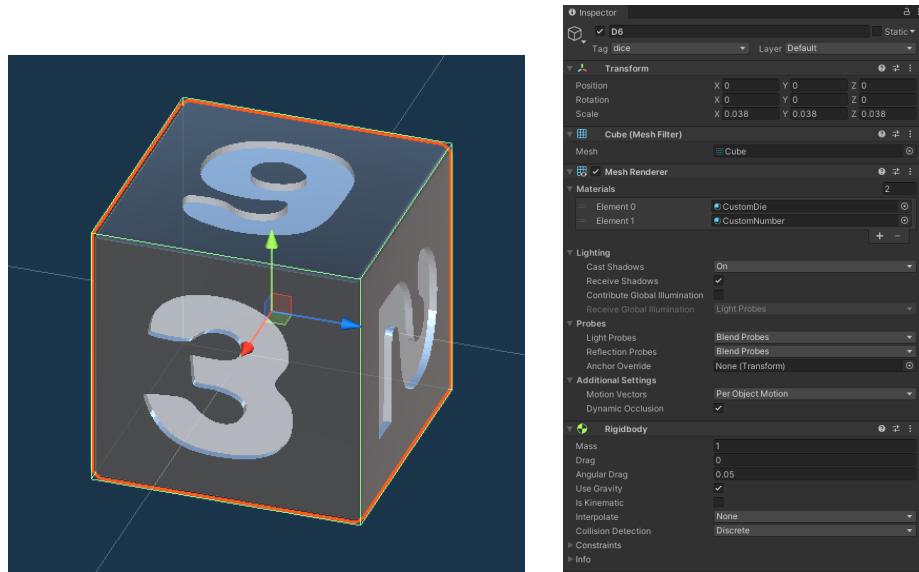


Figura 1.3: GameObject di un dado in ARDice

Come si può vedere infatti, il dado non è altro che un *GameObject* con determinati componenti: un *MeshFilter* e un *MeshRenderer*, dove il *MeshRenderer* prende la *Mesh* (ovvero la forma geometrica del dado) dal *MeshFilter* e la renderizza a schermo con i materiali che gli sono stati assegnati. Inoltre sono presenti un *RigidBody*, che rende il *GameObject* soggetto alle leggi della fisica, e un *MeshCollider*, il quale dà solidità al corpo e quindi gli permette di interagire con gli altri oggetti nella scena.

Ogni *GameObject* ha sempre un componente di tipo *Transform*, il quale è irremovibile e onnipresente. Esso è molto importante in quanto permette di mantenere informazioni inerenti a **posizione**, **rotazione** e **scala** sotto forma di vettori a 3 dimensioni (*Vector3*). Inoltre introduce il concetto di *parenting*, quindi la possibilità di creare *GameOject* innestati; funzionalità molto utile per la creazione di elementi complessi.

I valori di posizione, rotazione e scala della *Transform* sono infatti relative alla *Transform* del *GameObject* parent; mentre, se il *GameObject* non ha parent le proprietà sono misurate nel world space della scena di Unity.

Quindi come facilmente intuibile la potenza dei *GameObject* risede nella possibilità di poter rappresentare qualsiasi entità semplicemente aggiungendo Componenti allo stesso, inoltre è possibile creare anche Componenti personalizzati attraverso la scrittura di codice in C#.

La classe rappresentata da questi script, per essere considerata un componente, deve necessariamente estendere *MonoBehaviour*, che mette a disposizione una serie di metodi che lo sviluppatore potrà implementare per dare il comportamento desiderato al componente. Tra questi i più importanti sono:

- **Start()**: viene chiamato non appena lo script viene abilitato e prima di ogni chiamata del metodo Update;
- **Awake()**: viene chiamato appena l'istanza dello script viene caricata;
- **Update()**: viene chiamato ad ogni frame, se lo script è abilitato;
- **FixedUpdate()**: è una versione indipendente dal frame-rate del metodo Update, utilizzata generalmente per calcoli fisici.

Capitolo 2

AR Foundation

AR Foundation è un package di Unity che permette di lavorare con la realtà aumentata in multi-piattaforma. È un’interfaccia e in quanto tale necessita di un’implementazione per essere utilizzata. Per quanto concerne la nostra applicazione abbiamo fatto uso delle seguenti librerie:

- **ARCore** per Android
- **ARKit** per iOS

	ARCore	ARKit	Magic Leap	HoloLens
Device tracking	✓	✓	✓	✓
Plane tracking	✓	✓	✓	
Point clouds	✓	✓		
Anchors	✓	✓	✓	✓
Light estimation	✓	✓		
Environment probes	✓	✓		
Face tracking	✓	✓		
2D Image tracking	✓	✓	✓	
3D Object tracking		✓		
Meshing		✓	✓	✓
2D & 3D body tracking		✓		
Collaborative participants		✓		
Human segmentation		✓		
Raycast	✓	✓	✓	
Pass-through video	✓	✓		
Session management	✓	✓	✓	✓
Occlusion	✓	✓		

Figura 2.1: Tabella Feature per Platform AR Foundation 4.2.1

2.1 Architettura di AR Foundation

AR Foundation non è altro che un set di MonoBehaviour e API che si occupano di supportare molteplici incarichi. Esso si sviluppa su più livelli: i componenti principali (AR Components) vengono realizzati tramite vari sottosistemi (AR Subsystems), ognuno dei quali si occupa di una specifica funzionalità. Questi ultimi sono quindi il punto di connessione tra AR Foundation e le effettive implementazioni dei *provider*, facendo quindi da ponte tra Unity e i moduli degli *SDK AR "nativi"*.

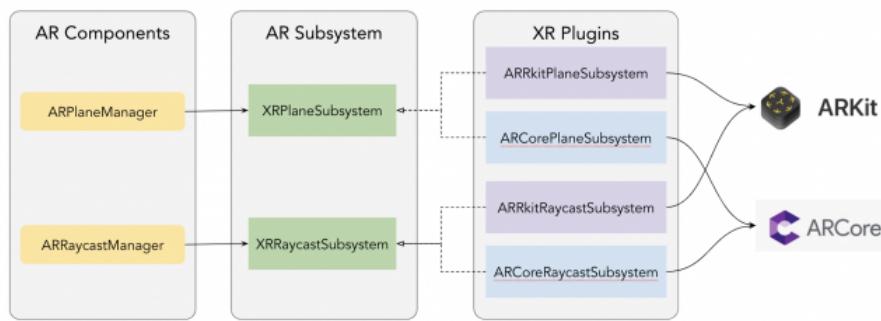


Figura 2.2: Architettura AR Foundation

I subsystem definiscono solo le interfacce dei metodi e non hanno nessun dettaglio implementativo. Questi sono invece scritti nelle differenti implementazioni specifiche per ogni dispositivo e il relativo *XR Plugin*. All'interno di queste librerie ogni specifica piattaforma sviluppa i metodi necessari per le funzionalità che essa può fornire all'utente, il tutto nel suo linguaggio nativo.

2.2 ARCore & ARKit

ARCore e ARKit sono dei software development kit realizzati rispettivamente da Google ed Apple per applicazioni Android ed iOS, che consentono esperienze di interazione con il mondo reale tramite l'utilizzo della realtà aumentata.

ARKit è un framework di sviluppo introdotto da Apple nel 2017 che consente agli sviluppatori di integrare motion features ai moduli della fotocamera per la produzione di applicazioni e giochi in AR. Unisce gli input dei sensori del dispositivo iOS con funzioni avanzate per processare la scena.

ARCore è un insieme di strumenti sviluppato e introdotto da Google nel 2018, diretto successore di Tango, che permette la realizzazione di applica-

zioni Android sfruttando la realtà aumentata in modo analogo ad ARKit, facendo uso di una vasta gamma di API per consentire allo smartphone di interagire con le informazioni fornite dall'ambiente circostante.

ARKit e ARCore hanno entrambi il proprio set di features, ma in un'applicazione cross-platform risultano di interesse ovviamente solo quelle condizionate. Di seguito le principali:

- **Motion tracking:** muovendo il dispositivo si fa uso di SLAM, un processo di localizzazione e mappatura simultanea per capire la sua posizione relativamente al mondo che lo circonda. Vengono rilevate caratteristiche distinte dalla fotocamera selezionando dei punti caratteristici e sfruttandoli per calcolare lo spostamento del dispositivo. L'informazione visiva è combinata alle misurazioni prodotte dall'IMU (unità di misura inerziale) per la stima della posa della telecamera, che comprende posizione e orientamento, allineando in tempo reale la fotocamera virtuale (che esegue il rendering 3D) con quella reale del dispositivo. L'immagine virtuale renderizzata può poi essere sovrapposta a quella catturata della fotocamera per dare l'illusione che gli oggetti virtuali facciano parte dell'ambiente.
- **Environmental understanding:** la comprensione dell'ambiente circostante è aggiornata e migliorata continuamente tramite la rilevazione di caratteristiche salienti come punti fissi e piani geometrici; in questo modo è possibile rilevare i confini di questi ultimi e rendere queste informazioni disponibili all'utente affinché possa usufruirne nell'applicazione.
- **Depth understanding:** viene realizzata una depth map contenente informazioni circa la distanza tra una superficie e un dato punto utilizzando la fotocamera del dispositivo. Queste informazioni vengono sfruttate per posizionare in modo migliore gli oggetti virtuali nella scena.
- **Light estimation:** è possibile fare una stima dell'illuminazione dell'ambiente per cercare di replicarla nell'ambiente di gioco, aumentando quindi la qualità della simulazione.

Capitolo 3

AR Dice

AR Dice è un'applicazione mobile sviluppata su *Unity* per dispositivi Android e iOS. Essa utilizza il riconoscimento di superfici orizzontali e verticali per creare un *virtual environment* il più simile possibile alla realtà. Questo consente all'utente di interagirvi con oggetti virtuali, sia dal punto di vista delle collisioni che delle occlusioni ottiche.

Oltre alle due modalità di lancio dei dadi l'utente ha anche la possibilità di creare e gestire il proprio tavolo da gioco definendone perimetro, altitudine, ed inserendovi oggetti virtuali. Considerando che un affidabile riconoscimento di oggetti non è ancora effettuabile dalla maggior parte dei dispositivi mobile in commercio, per il momento nello sviluppo del prototipo abbiamo scelto di affidarci alla *image recognition* di alcune **tessere** per poter aggiungere al mondo di gioco degli oggetti virtuali in modo dinamico, con dimensioni scalabili, al fine di simulare quelli reali e contro i quali i dadi nella scena di gioco possano effettivamente collidere.

I dadi utilizzati all'interno dell'applicazione sono stati da noi modellati tramite Blender, permettendoci di realizzarli in base alle nostre necessità sia per dimensioni che in aspetto, separando quindi a livello di rendering il materiale del corpo da quello del numero per garantirci migliore leggibilità del risultato e personalizzazione.

Con le attuali funzionalità fornite dall'applicazione i requisiti minimi sono coincidenti con quelli di ARCore e ARKit, quindi rispettivamente *Android Nougat* (API Level 24) e *iOS 11*.

3.1 AR Foundation: componenti impiegati

Di seguito verranno brevemente analizzati i principali componenti di AR-Foundation utilizzati all'interno di ARDice per realizzare le sue funzionalità chiave.

3.1.1 ARSession

ARSession è il componente che controlla il ciclo di vita e le opzioni per la sessione di realtà aumentata. È possibile mantenere attiva una sola sessione AR e se disabilitata il sistema smetterà di tracciare le caratteristiche dell'ambiente.

Trattandosi dunque di una sessione, tutte le caratteristiche tracciate, i piani disegnati e ogni altro oggetto inserito nella scena saranno relativi ad essa; di conseguenza verranno eliminati una volta chiusa l'applicazione.

3.1.2 ARSessionOrigin & ARCamera

ARSessionOrigin è il componente fondamentale di ogni applicativo AR sviluppato tramite ARFoundation. Esso contiene la Camera, nello specifico **ARCamera**, ed è il container per ogni GameObject inerente al mondo AR, ad esempio piani o point clouds.

Posizione e rotazione della Camera dipendono dal dispositivo, quindi sono inizializzati a 0 durante il caricamento dell'applicazione, e di conseguenza i tratti dell'ambiente rilevati verranno posizionati relativamente ad essa nel mondo di gioco. Chiamiamo lo spazio relativo al dispositivo "*session/device space*".

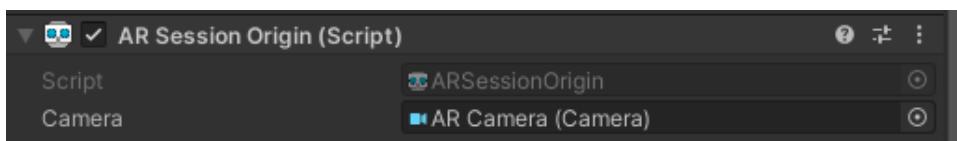


Figura 3.1: AR Session Origin

Lo scopo di ARSessionOrigin è inoltre di convertire lo spazio della sessione nello spazio di gioco di Unity, per questo motivo è importante che la Camera e le caratteristiche rilevate dell'ambiente siano nello stesso spazio.

3.1.3 ARPlaneManager

Prima di iniziare a spiegare i *Manager Components* di ARFoundation, è necessario precisare cosa questa libreria definisca come *trackable*. Un trackable è qualsiasi cosa possa essere rilevata e tracciata nel mondo reale; quindi ad esempio piani, nuvole di punti, anchors, facce, immagini e oggetti 3D.

Ogni trackable ha un proprio *trackable manager* che deve risiedere nello stesso GameObject del ARSessionOrigin, necessario in quanto ARSessionOrigin definisce la Transform a cui sono relativi tutti i trackable rilevati. Inoltre i trackable manager utilizzano la ARSessionOrigin per posizionare il trackable rilevato nella corretta posizione all'interno della scena di Unity.

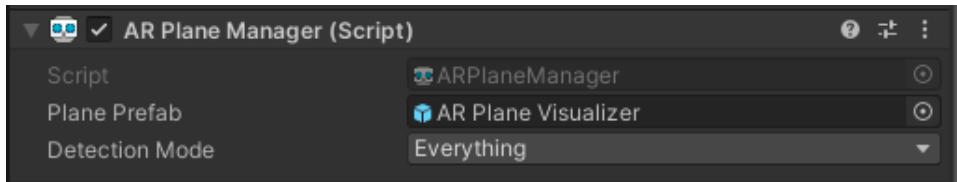


Figura 3.2: AR Plane Manager

ARPlaneManager è il componente che permette di costruire dei GameObject di tipo **ARPlane** in prossimità delle superfici verticali ed orizzontali rilevate dalla videocamera nell'ambiente reale.

Un ARPlane non è altro che una superficie caratterizzata da una *Pose*, classe di ARFoundation contenente informazioni su posizione e rotazione di un GameObject, una dimensione e dei *boundary points*, ovvero punti che rappresentano i vertici del piano stesso.

Come si può evincere dalla Fig. 3.2 è possibile selezionare la *detection mode*; quindi specificare se rilevare superfici orizzontali, verticali oppure entrambe.

Non appena un piano viene individuato, il PlaneManager ne istanzia il GameObject. Più nel dettaglio, una volta per frame viene invocato l'evento *planesChanged*, il quale aggiorna la lista di piani che sono stati aggiunti, rimossi o modificati rispetto al frame precedente.

Inoltre, quando un piano viene modificato, significa che i suoi boundary points vengono modificati. ARPlaneManager effettua questa modifica attraverso l'evento *boundaryChanged*, il quale viene invocato non appena almeno un vertice viene modificato.

3.1.4 ARRaycastManager

Per Raycasting in Unity s'intende quando un *Ray*, GameObject definito da un'origine e una direzione (quindi un vettore), interseca un qualsiasi altro GameObject presente nella scena.

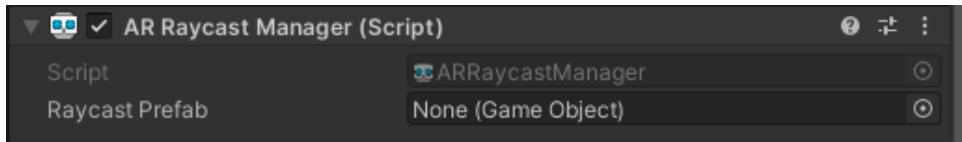


Figura 3.3: AR Raycast Manager

ARRaycastManager non differisce molto dall'interfaccia di Raycasting del modulo Physics di Unity. La differenza principale sta nel tipo di oggetti con i quali il Ray si scontra; infatti il ARRaycastManager determina quando il Ray colpisce ARGameObject, ad esempio ARPlane oppure un elemento appartenente ad una nuvola di punti.

3.1.5 AROcclusionManager

L'**AROcclusionManager** è il componente che si occupa di gestire le occlusioni visive degli oggetti, nascondendoli dietro le superfici rilevate quando queste vengono sovrapposte ad essi.

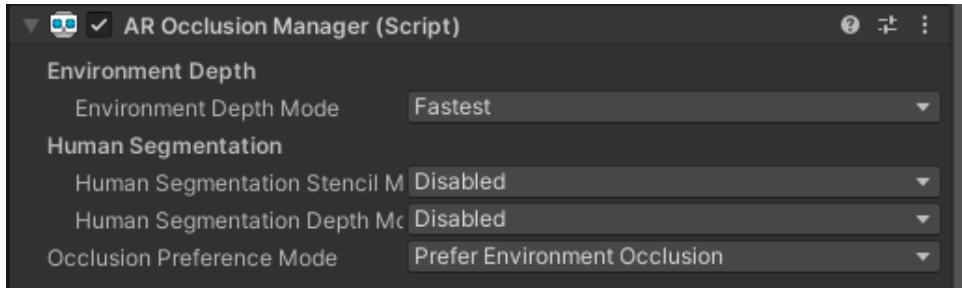


Figura 3.4: AR Occlusion Manager

Ovviamente queste occlusioni dipendono dalla stima della profondità che viene fatta dal Plane Manager, quindi la precisione di queste occlusioni dipende da esso e dalla tecnologia fornita dal dispositivo in utilizzo. Nuovi dispositivi forniti di LiDAR scanner potranno ovviamente produrre previsioni e risultati migliori, previa aggiunta dei componenti necessari per il suo corretto funzionamento, ovvero **ARMeshManager**.

3.1.6 ARTrackedImageManager

ARTrackedImageManager è un componente che si occupa della creazione di un GameObject nella scena virtuale per ciascuna immagine rilevata tramite la fotocamera nell'ambiente reale. Affinché il componente sia in grado di rilevare un'immagine è necessario che questa sia inserita preventivamente in una libreria che vi andrà associata.

La Reference Library richiede che ciascuna immagine abbia un livello di complessità sufficiente affinché sia possibile effettuare il tracciamento, inoltre è prevista la possibilità di specificare le dimensioni per ciascuna di esse per migliorare l'accuratezza della rilevazione.

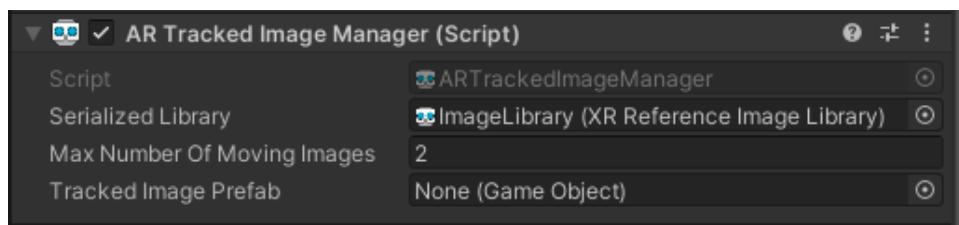


Figura 3.5: AR Tracked Image Manager

3.2 Struttura Software

La nostra applicazione è sviluppata interamente all'interno di una sola *MainScene*. Questa scelta deriva dal fatto che quasi ogni elemento al suo interno è indispensabile al corretto funzionamento di altri componenti. Inoltre, data la possibilità di poter disattivare o attivare oggetti tramite la funzione *SetActive(bool)* possiamo facilmente gestire quali debbano essere le funzionalità e gli elementi a schermo necessari in ogni dato momento di esecuzione.

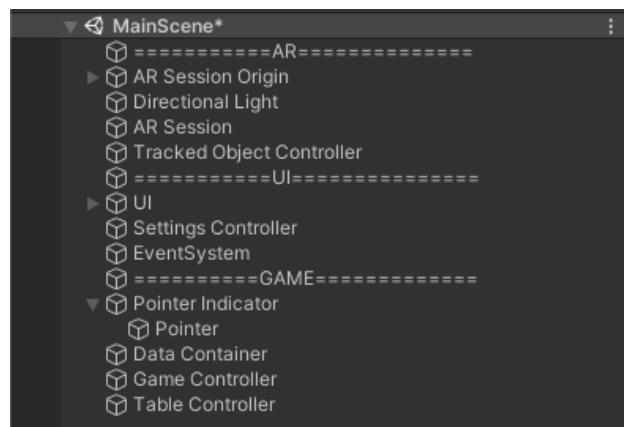


Figura 3.6: Main Scene di AR Dice

Gli elementi cardine di AR Dice sono quindi: *Data Container*, *Game Controller*, *Table Controller*, *Image Tracking Controller*, *Settings Controller* e *Persistance Controller*.

3.2.1 Data Container

Il **Data Container** è un componente che, sfruttando il pattern *Singleton*, ci permette di gestire tutti quegli oggetti che sono utilizzati da più script. Questi vengono pre-caricati al suo interno e, all'occorrenza, ne viene restituita l'istanza all'utilizzatore.

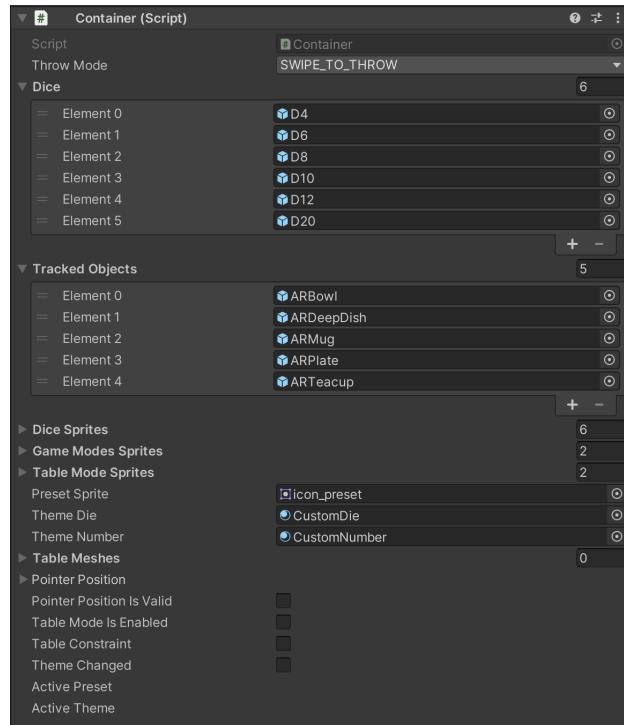


Figura 3.7: Data Container

3.2.2 Game Controller

Il **Game Controller** rappresenta il core del normale utilizzo di ARDice, infatti è il componente che permette di gestire il lancio di dadi nel mondo AR e farli interagire con l'ambiente circostante.

Prima di spiegare nel dettaglio la classe è necessario descrivere cosa s'intenda per **GameMode**. Essa rappresenta la principale modalità di utilizzo dell'applicazione, attraverso la quale vengono lanciati i dadi e mostrato il risultato del dado.

In GameMode sono presenti due tipologie di lancio dei dadi, perfettamente intercambiabili l'una all'altra in qualsiasi momento dall'utente, esse sono:

- **Falling Dice:** in questa tipologia il dado viene lanciato in prossimità di un puntatore posizionato nell'ambiente AR attraverso una stima di profondità effettuata dal ARRaycastManager, facendo tracking sui piani;

- **Swipe to throw:** tipologia nella quale il dado correntemente selezionato è visibile a schermo ed è possibile lanciarlo attraverso una gestione di swipe sullo stesso.

Oltre che a gestire il lancio di dadi, questa classe si occupa anche della gestione dell'interfaccia utente inerente alla modalità di gioco, e quindi controlla gli eventi inerenti a:

- cambio di tipologia di lancio;
- passaggio dalla GameMode alla **TableMode** (modalità che verrà analizzata nel paragrafo successivo) e viceversa;
- cambio del dado da lanciare;

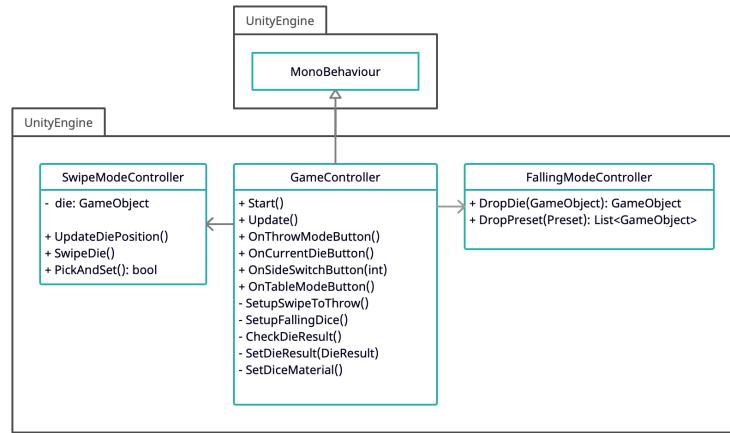


Figura 3.8: Game Controller

Come mostrato in Fig. 3.8 il componente che realizza il GameController è l'omonima classe, la quale estende *MonoBehaviour*. Inoltre per realizzare le due modalità di lancio sono state create due classi accessorie, rispettivamente **SwipeModeController** e **FallingModeController**. Queste due classi non estendono *MonoBehaviour* ma semplicemente espongono dei metodi che GameController utilizza per lanciare i dadi.

Falling Dice

Prima di analizzare FallingModeController è necessario spiegare il posizionamento del puntatore. Il puntatore non è altro che un *GameObject*, inizialmente posizionato nell'origine, al quale è stato "attaccato" uno script Mo-

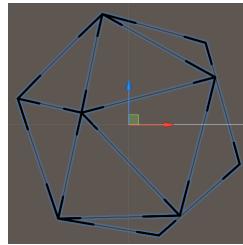


Figura 3.9: Puntatore

noBehaviour che permette di posizionarlo all'interno della scena attraverso stime di profondità effettuate dal ARRaycastManager, nel dettaglio:

```

1  private void UpdatePointerPose() {
2      var screenCenter = Camera.current.
3          ↪ViewportToScreenPoint(new Vector3(0.5f, 0.5f,
4              ↪0.5f));
5      var hits = new List<ARRaycastHit>();
6
7      raycastManager.Raycast(screenCenter, hits,
8          ↪TrackableType.Planes);
9
10     Container.instance.pointerPositionIsValid = hits.
11         ↪Count > 0;
12
13     if (Container.instance.pointerPositionIsValid) {
14         Container.instance.pointerPosition = hits[0].pose
15             ↪;
16
17         Vector3 cameraForward = Camera.current.transform.
18             ↪forward;
19         Vector3 cameraBearing = new Vector3(cameraForward
20             ↪.x, 0, cameraForward.z).normalized;
21
22         Container.instance.pointerPosition.rotation =
23             ↪Quaternion.LookRotation(cameraBearing);
24     }
25 }
```

Listing 3.1: Metodo UpdatePointerPose()

La stima di profondità viene eseguita nel metodo raycastManager.Raycast(), il quale vuole in ingresso:

- *Vector2*: il punto dello schermo da cui far partire il raycast;
- *List<ARRaycastHit>*: lista che viene popolata non appena il ray colpisce un piano;
- *TrackableType*: ovvero il tipo di trackable sul quale fare raycasting; nel caso di ARDice è TrackableType.Planes.

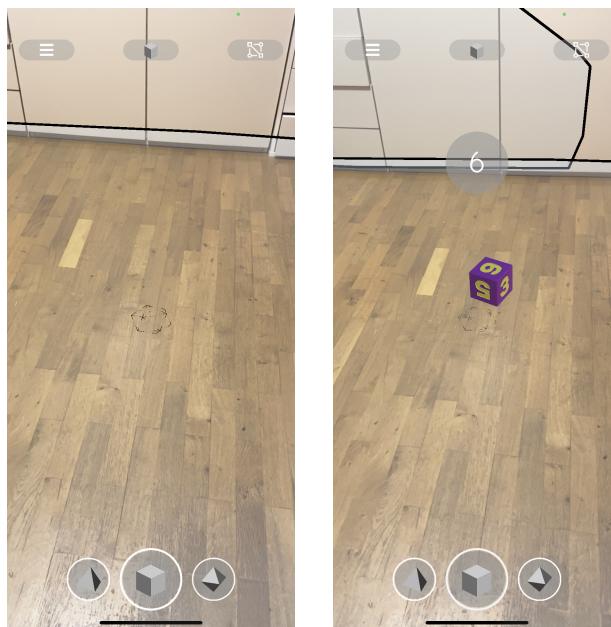


Figura 3.10: Esempio di lancio tramite FallingDice

Ora che è stato introdotto il posizionamento del puntatore è possibile descrivere il funzionamento di FallingDice. A differenza della SwipeToThrow, nella quale c’è un’interazione con il dado, in FallingDice il dado viene lanciato alla pressione di un tasto, per la precisione il tasto al centro riportante l’immagine del dado correntemente selezionato.

È importante notare che, per scelte progettuali, FallingDice è l’unica tipologia di lancio nella quale viene data la possibilità di lanciare più dadi contemporaneamente. Questa scelta è dovuta dal fatto che in modalità SwipeToThrow è estremamente complicato gestire l’anteprima di multipli dadi e di tipi diversi, quindi si è scelto di confinare questa possibilità in FallingDice, dove non è necessaria alcuna anteprima dei dadi da lanciare.

Quindi, per dar la possibilità di lanciare più dadi contemporaneamente è stata creata una classe apposita chiamata **Preset** la quale mantiene solamente una lista di interi utilizzata per sapere quanti dadi e di quale tipo dovranno essere lanciati. Un utilizzo più approfondito di questa classe verrà analizzato nell’ultima sezione di questo capitolo.

La classe *FallingModeController* espone due metodi:

- *DropDie(GameObject)*: prende in ingresso il *GameObject* del dado da lanciare, lo istanzia e lo ritorna;

- DropPreset(Preset): prende in ingresso un Preset, istanzia il numero e il tipo di dadi indicati dal preset e ritorna la lista di GameObject costituita dai dadi istanziati.

Nello specifico:

```

1 public GameObject DropDie(GameObject currentDie) {
2     GameObject res = null;
3     float y = Container.instance.pointerPosition.y + 1.5f
4         ↵;
5
6     Vector3 v = new Vector3(Container.instance.
7         ↵pointerPosition.position.x, y, Container.
8         ↵instance.pointerPosition.position.z);
9     Vector3 torque = new Vector3();
10
11    torque.x = Random.Range(-200, 200);
12    torque.y = Random.Range(-200, 200);
13    torque.z = Random.Range(-200, 200);
14
15    res = Instantiate(currentDie, v, Container.instance.
16        ↵pointerPosition.rotation);
17
18    Rigidbody rb = res.GetComponent<Rigidbody>();
19    rb.isKinematic = false;
20    rb.AddTorque(torque);
21
22    return res;
23 }
```

Listing 3.2: Metodo DropDie()

Da notare come nella linea 4 del codice di DropDie() viene creato un Vector3 con le componenti x e z ottenute dalla posizione del puntatore, mentre la y impostata ad 1.5m sopra il puntatore, in quanto un'unità in Unity è 1 metro nel mondo AR, rendendo così possibile la caduta del dado fino alla posizione desiderata.

Il codice della DropPreset() è uguale a DropDie(), l'unica differenza risiede nell'utilizzo di Preset e quindi in una necessità di effettuare un ciclo su ogni tipo di dado.

Come detto all'inizio, in FallingDice, il lancio del dado avviene alla pressione di un bottone, al quale è associato un evento attraverso il metodo *OnCurrentDieButton()*, nel dettaglio:

```

1 public void OnCurrentDieButton() {
2     if (throwMode == ThrowMode.SWIPE_TO_THROW) {
3         ...
4     } else if (throwMode == ThrowMode.FALLING) {
5         if (instantiatedDie != null) {
6             Destroy(instantiatedDie);
7         }
8 }
```

```

9   // lancio di preset
10  if (currentDie == dice.Count) {
11      if(presetInstantiatedDice.Count > 0) {
12          for(int i = presetInstantiatedDice.Count
13              →- 1; i >= 0; i--) {
14              Destroy(presetInstantiatedDice[i]);
15              presetInstantiatedDice.RemoveAt(i);
16          }
17      }
18      presetInstantiatedDice =
19          →fallingModeController.DropPreset(
20              →Container.instance.activePreset);
21  } else {
22      instantiatedDie = fallingModeController.
23          →DropDie(dice[currentDie]);
}
}

```

Listing 3.3: Metodo OnCurrentDieButton()

Swipe To Throw

Figura 3.11: GameObject di un dado in ARDice

La tipologia di lancio SwipeToThrow, come anticipato in precedenza, permette di lanciare il dado attraverso una gestione di swipe sullo stesso, come illustrato in Fig. 3.11. Di default, prima di essere lanciato, il dado

viene mantenuto tramite script in una posizione prefissata in primo piano, dipendentemente dalla *Transform* della ARCamera.

Il riconoscimento della gestione di swipe, all'interno di GameController, avviene grazie alla Update() tramite il metodo SwipeDie() di SwipeModeController il quale, non appena riconosce la gestione, applica una forza d'impulso al GameObject del dado direttamente proporzionale al modulo del vettore disegnato con la gestione di swipe. Quindi il dado compirà una traiettoria a parabola fino a collidere con i piani riconosciuti e costruiti dal ARPlaneManager.

Lo swipe del dado è gestito dal seguente metodo:

```

1  public void SwipeDie() {
2      Vector3 startPosition;
3      Vector3 endPosition;
4      float touchStartTime;
5      float touchEndTime;
6      float timeInterval;
7
8      if (Input.touchCount > 0) {
9          Touch touch = Input.GetTouch(0);
10
11         if (touch.phase == TouchPhase.Began) {
12             startPosition = touch.position;
13             touchStartTime = Time.time;
14
15             Ray ray = Camera.current.ScreenPointToRay(
16                 startPosition);
17             RaycastHit hitObject;
18
19             if (Physics.Raycast(ray, out hitObject)) {
20                 if (hitObject.transform.tag.Equals("dice"))
21                     {
22                         onTouchHold = true;
23                         rigidbody.isKinematic = true;
24                     }
25             }
26
27             if (onTouchHold && touch.phase == TouchPhase.
28                 Ended) {
29                 // Gesture recognized
30
31                 onTouchHold = false;
32                 touchEndTime = Time.time;
33                 timeInterval = touchEndTime - touchStartTime;
34                 endPosition = touch.position;
35                 Vector3 direction = endPosition -
36                     startPosition;
37
38                 // Random Torque
39                 torque.x = Random.Range(-200, 200);
40                 torque.y = Random.Range(-200, 200);
41                 torque.z = Random.Range(-200, 200);
42
43                 // Camera Rotation and Reset
44             }
45         }
46     }
47 }
```

```
41     float xCameraRotation = Camera.current.  
42         ↪transform.forward.x;  
43     float zCameraRotation = Camera.current.  
44         ↪transform.forward.z;  
45     float xForce = (throwForceZ / timeInterval) *  
46         ↪xCameraRotation;  
47     float zForce = (throwForceZ / timeInterval) *  
48         ↪zCameraRotation;  
49     float yForce = throwForceY * direction.y;  
50  
51     // Impulse force on gameobject  
52     rigidbody.isKinematic = false;  
53     rigidbody.AddForce(xForce, yForce, zForce);  
54     rigidbody.AddTorque(torque);  
55 }  
56 }
```

Listing 3.4: Método SwipeDie()

Inoltre, una volta lanciato il dado, è possibile recuperarlo per poterlo rilanciare semplicemente facendo un tap sullo stesso attraverso il metodo PickAndSet() di SwipeModeController; il quale riposiziona il dado in posizione di lancio.

3.2.3 Table Controller

In questa sezione verrà analizzato ciò che la **TableMode** rappresenta.

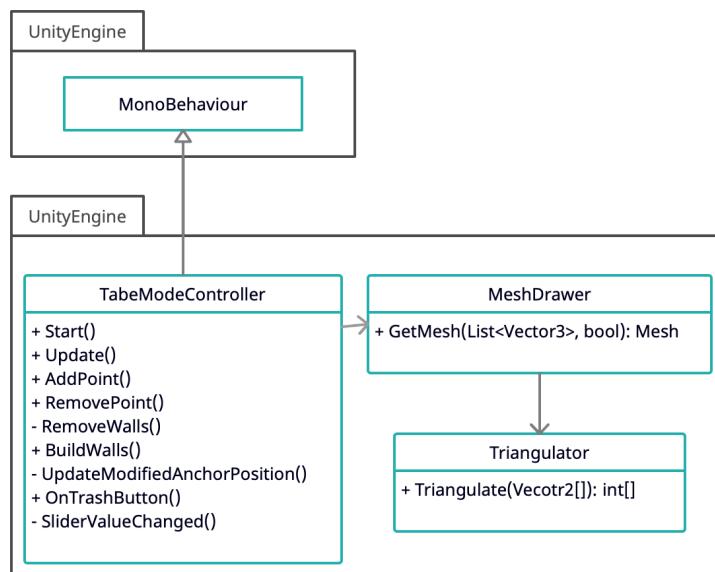


Figura 3.12: Table Controller

TableMode è la modalità nella quale è possibile costruire un tavolo personalizzato vertice per vertice. La necessità di questa modalità sorge quando è presente una superficie molto complessa con la quale ARPlaneManager fatica a costruire un piano preciso.

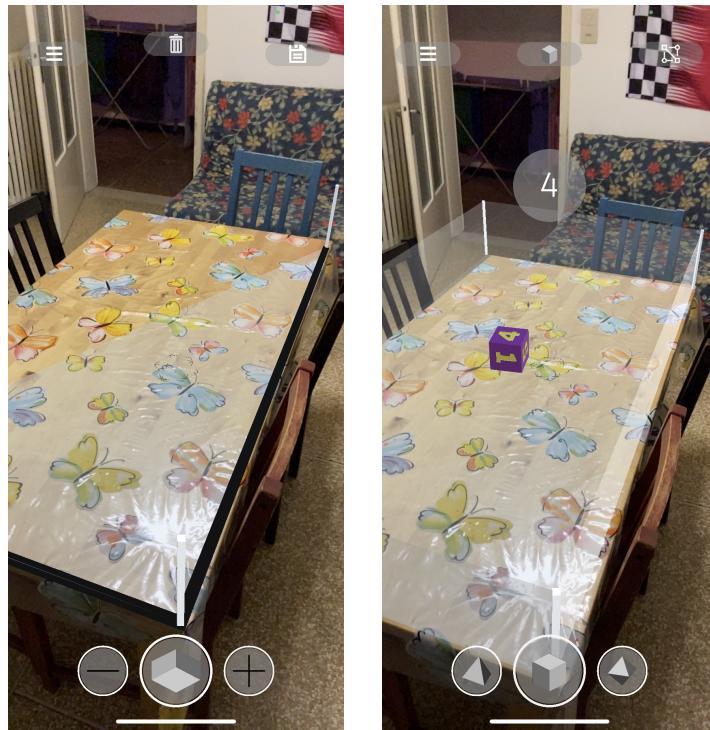


Figura 3.13: Costruzione tavolo attraverso la TableMode

In TableMode è di nuovo fondamentale l'utilizzo del puntatore analizzato nella FallingDice. In questa modalità il puntatore serve per posizionare i vertici del tavolo. Man mano che vengono aggiunti vertici il TableController, essa mostra un'anteprima della superficie che si sta costruendo attraverso la generazione di Mesh a partire dalle coordinate dei vertici. Ciò è possibile attraverso la classe **MeshDrawer** la quale, a partire da una lista di Vector3, ritorna un oggetto di tipo Mesh. Essa verrà poi assegnata al *MeshFilter* di TableController e renderizzata dal *MeshRenderer*.

La classe MeshDrawer permette rappresentare correttamente la Mesh tramite l'utilizzo di una classe ausiliaria chiamata Triangulator che fornisce il corretto l'ordine nel quale i vertici devono essere utilizzati per disegnare correttamente la Mesh.

Andando più a fondo nel codice, questo è il metodo per aggiungere vertici:

```

1 public void AddPoint() {
2     if (pointerPositionIsValid && !selected) {
3         Vector3 p = Container.pointerPosition.position;
4         Quaternion r = Container.pointerPosition.rotation
5             →;
6
7         if (vertices.Count == 0) {
8             fixedY = p.y;
9             meshPrefabClone = Instantiate(meshPrefab);
10            meshes.Add(meshPrefabClone);
11        }
12
13        p.y = fixedY;
14        GameObject g = Instantiate(anchorPrefab, p, );
15        g.transform.GetChild(0).name = vertices.Count.
16            →ToString();
17        vertices.Add(g);
18
19        if (vertices.Count > 2) {
20            List<Vector3> points = vertices.Select(v => v
21                →.transform.position).ToList();
22            Mesh mesh = meshDrawer.GetMesh(points);
23            meshes[0].GetComponent<MeshFilter>().
24                →sharedMesh = mesh;
25            meshes[0].GetComponent<MeshCollider>().
26                →sharedMesh = mesh;
27        }
28
29    }
30}

```

Listing 3.5: Metodo AddPoint()

Nel dettaglio:

- *meshPrefabClone*: rappresenta il GameObject contenente la Mesh del piano;
- *fixedY*: è un float che viene memorizzato alla costruzione del primo vertice, viene utilizzato in seguito per forzare tutti i vertici ad essere alla stessa Y e quindi creare un piano perfettamente orizzontale;
- *vertices*: è una lista di GameObject che rappresentano i vertici del tavolo, visibili a schermo attraverso dei sottili parallelepipedi.

Ovviamente è data anche la possibilità di rimuovere vertici già costruiti, attraverso il metodo *RemovePoint()*, seguito da un ricalcolo della Mesh.

Inoltre, visto che ora come ora il tavolo costruito attraverso questa modalità è solo un piano, e quindi i dadi rotolando potrebbero cadere al di fuori di esso, è stata aggiunta la costruzione di muri attorno al suo perimetro, impedendo quindi al dado di cadere al di fuori del tavolo.

La loro costruzione è affidata al metodo *BuildWalls()* il quale per ogni coppia di punti ne costruisce altri due con le stesse coordinate tranne la y, la quale viene alzata di 0.2f. Successivamente per ogni quaterna di punti rappresentanti il singolo muro viene costruita la Mesh.

In aggiunta, in quanto la stima effettuata dal ARRaycastManager nel posizionamento del puntatore non è molto precisa, si è deciso di dare la possibilità di alzare ed abbassare il piano lungo l'asse y. Ciò viene gestito da uno slider e dal metodo *SliderValueChanged()*

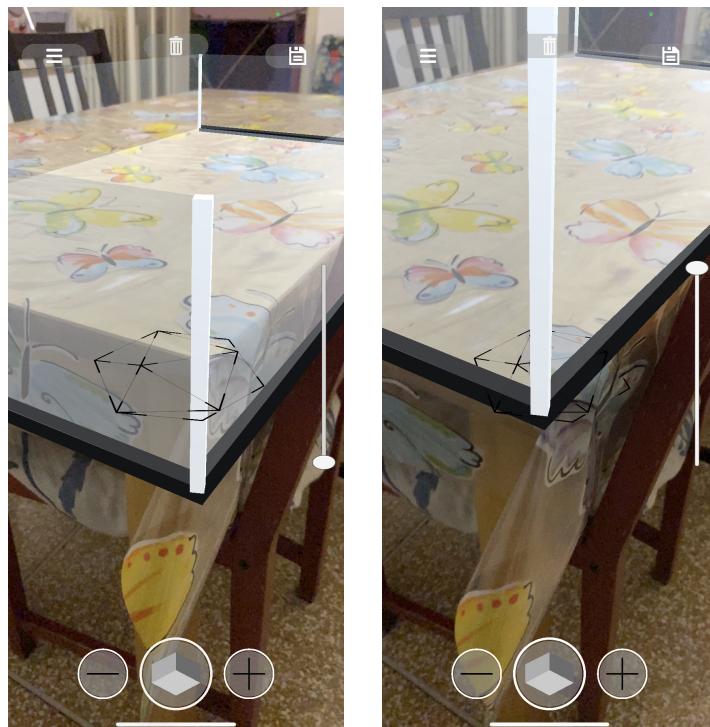


Figura 3.14: Dimostrazione funzionamento slider per modifica dell'altezza del tavolo

Per esaustività è necessario spiegare l'interfaccia utente della TableMode, facendo riferimento all'immagine 3.14, partendo dai tre tasti nella parte bassa dello schermo:

- il tasto (+) permette di aggiungere vertici;
- il tasto (-) permette di rimuovere il vertice selezionato;
- il tasto centrale consente di costruire i muri a partire dai vertici costruiti.

Invece facendo riferimento ai tre tasti superiori:

- il tasto centrale permette di eliminare i vertici ed i muri costruiti;
- il tasto a destra consente di salvare il tavolo correntemente disegnato e quindi uscire dalla TableMode per tornare in GameMode;
- il tasto a sinistra apre il Settings Menù, il quale verrà analizzato successivamente.

3.2.4 Image Tracking Controller

ImageTrackingController è la classe che gestisce la modalità di inserimento degli oggetti tramite tessere nella scena virtuale.

All'avvio dell'applicazione, nel metodo Awake(), viene istanziata una lista con i GameObject degli oggetti di cui è previsto il possibile inserimento in scena. Dopodiché, non appena vengono rilevate una o più tessere dalla fotocamera, la classe vi colloca sopra il corrispettivo GameObject, aggiornandone posizione e rotazione costantemente.

```

1  private void Awake() {
2      trackedImageManager = FindObjectOfType<
3          >ARTrackedImageManager>();
4      arRaycastManager = GetComponent<ARRaycastManager>();
5
6      foreach (GameObject prefab in placedPrefabs) {
7          GameObject newPrefab = Instantiate(prefab,
8              <Vector3>.zero, Quaternion.identity);
9          newPrefab.SetActive(false);
10         newPrefab.name = prefab.name;
11         spawnedPrefabs.Add(prefab.name, newPrefab);
12         prefabsList.Add(prefab.name, newPrefab);
13     }
14 }
```

Il riconoscimento dell'immagine avviene attraverso una **ReferenceLibrary**, la quale deve essere preventivamente popolata dalla lista delle immagini da riconoscere e dal rispettivo GameObject da costruirvi sopra, e quindi fornita al ARTrackedImageManager.

Inoltre nella classe ImageTrackingController è stato predisposto un metodo chiamato ImageChanged() il quale, in base agli eventi generati da ARTrackedImageManager del tipo ARTrackedImagesChangedEventArgs, gestisce l'aggiunta, la modifica e lo spostamento dei GameObject.

Nel dettaglio:

```

1  private void ImageChanged(ARTrackedImagesChangedEventArgs
2      <eventArgs>) {
3      foreach (ARTrackedImage trackedImage in eventArgs.
4          <added>) {
5          UpdateImage(trackedImage);
6          spawnedPrefabs.Add(trackedImage.name, prefabsList
7              <[trackedImage.name]>);
8      }
9
10     foreach (ARTrackedImage trackedImage in eventArgs.
11         <updated>) {
12         UpdateImage(trackedImage);
13     }
14 }
```

```

10
11     foreach (ARTrackedImage trackedImage in eventArgs.
12         ↪removed) {
13         Destroy(spawnedPrefabs[trackedImage.name]);
14         spawnedPrefabs.Remove(trackedImage.name);
15     }
16
17     private void UpdateImage(ARTrackedImage trackedImage) {
18         string name = trackedImage.referenceImage.name;
19         Vector3 position = trackedImage.transform.position;
20         Quaternion rotation = trackedImage.transform.rotation
21         ↪;
22         GameObject prefab = spawnedPrefabs[name];
23         prefab.SetActive(true);
24         prefab.transform.position = position;
25         prefab.transform.rotation = rotation;
}

```

Inoltre è stata predisposta anche una classe, chiamata TrackedGameObjectController che permette di fissare in posizione dei cloni per ciascun oggetto costruito sulle tessere tramite un tap su schermo. È inoltre possibile selezionare ciascuno di questi ultimi singolarmente per modificarne le dimensioni ed adattarli fedelmente ai corrispettivi oggetti presenti nell'ambiente reale che essi rappresentano. Questo viene permesso grazie a due slider, uno per la base e uno per l'altezza.

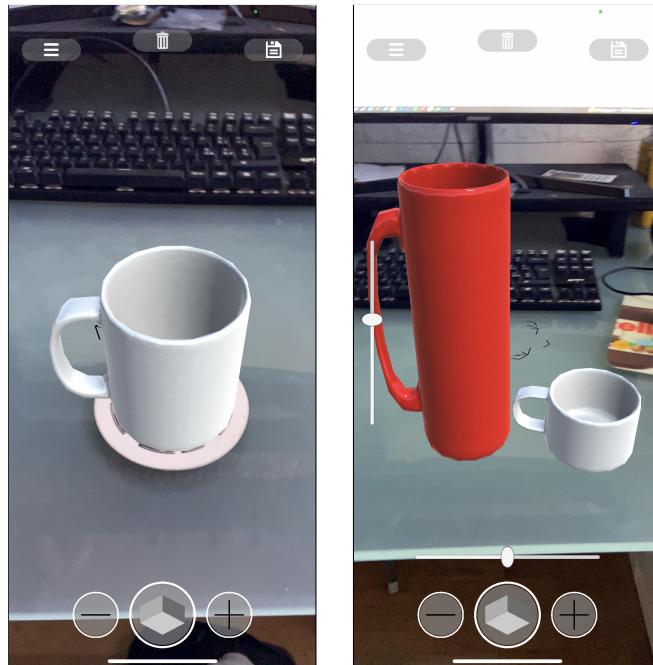


Figura 3.15: Esempio di riconoscimento immagini e modifica della dimensione del gameobject riconosciuto

3.2.5 Settings Controller e Persistance Controller

Settings Controller è il componente che si occupa della gestione e della persistenza delle due impostazioni principali dell'applicazione: *Presets* e *Themes*.

Tramite un menù a scomparsa è possibile scegliere se accedere ad una delle schede di selezione oppure alla pagina di Help, che permette di visualizzare una breve guida per l'utente.

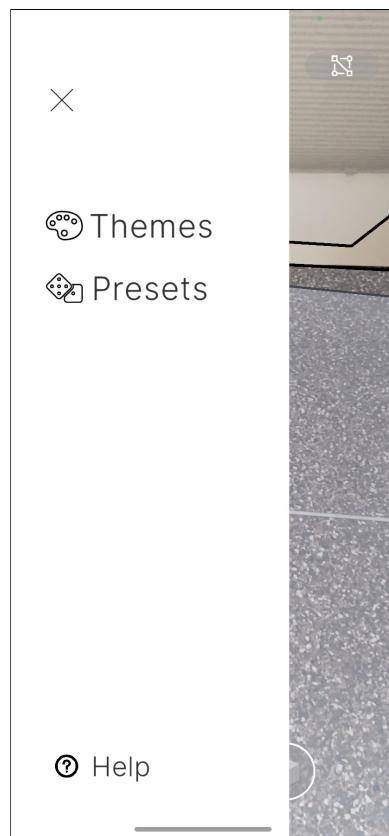


Figura 3.16: Menù a scomparsa

In entrambi i menu è possibile fare un semplice *Tap* per selezionare un elemento oppure un *Tap and Hold* per entrare nel relativo menu di modifica. Sia nei temi che nei preset è possibile selezionare uno e un solo elemento, senza possibilità di deselectarlo.

La modifica del tema permette, tramite una *color wheel*, di selezionare il colore desiderato. Toccando la *preview* del dado è possibile spostarsi tra la modifica del colore del corpo e quello dei numeri.

Una volta modificato basterà selezionarlo per cambiare il colore del *Material* dei dadi di gioco.

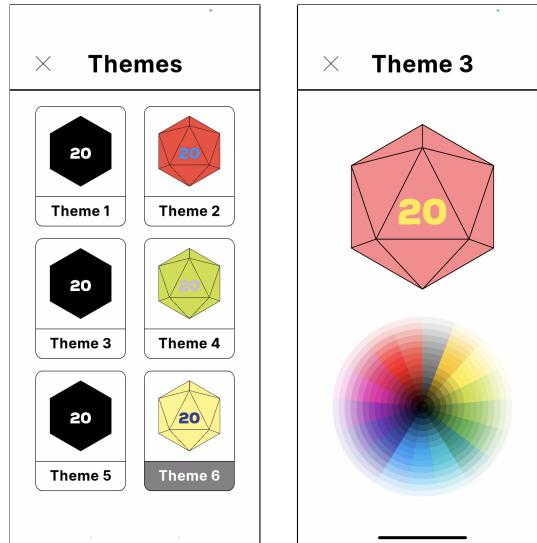


Figura 3.17: Lista di selezione dei temi e personalizzazione

Per quanto riguarda la modifica dei preset, viene permesso all’utente di modificare il numero dei dadi da lanciare per ogni tipologia. Al momento viene permesso di inserire fino a 25 dadi al loro interno, ma è facilmente possibile modificare questa impostazione.

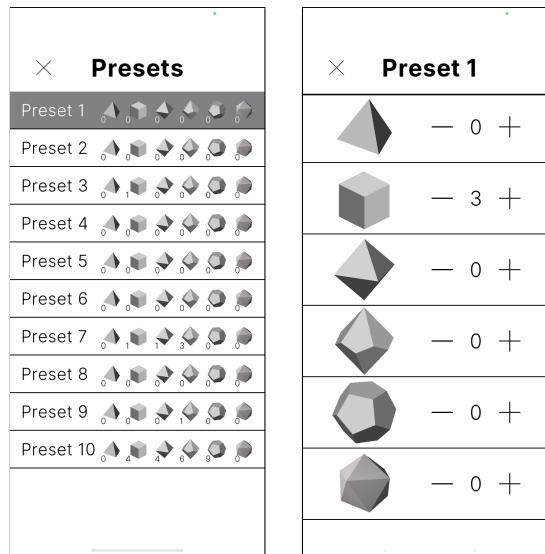


Figura 3.18: Lista di selezione dei preset e personalizzazione

Settings Controller si occupa inoltre, ogni volta che avviene una selezione

o una modifica, di chiamare **Persistance Controller**. Esso non è altro che una classe che si occupa di caricare e salvare i file di preset, temi e selezioni correntemente effettuate.

Il caricamento di questi file avviene all'avvio dell'applicazione tramite una chiamata di SettingsController, che quindi carica sul Container le informazioni necessarie agli altri componenti.

Al momento esso gestisce un numero limitato e fisso di preset e di temi, in particolare 10 preset e 6 temi. Qualora in un secondo momento si voglia permettere all'utente di aggiungerne di numero i metodi necessari per rendere queste quantità variabili sono già presenti.

Theme e *Preset* sono due classi con attributo *Serializable* che, a scapito di restrizioni sulle tipologie di dato che esse possano contenere, ci permette di poterle facilmente salvare in file binari tramite stream.

```

1 [System.Serializable] public class Preset {
2     private int _LIMIT = 25;
3     private int current;
4     private List<int> quantities;
5
6     public Preset() {
7         quantities = new List<int>();
8         // Order in list is d4, d6, d8, d10, d12, d20
9         for(int i = 0; i < 6; i++)
10             quantities.Add(0);
11
12         current = 0;
13     }
14     [...]
15 }
```

Listing 3.6: Preset Class

```

1 [System.Serializable] public class Theme {
2     private float _dieR, _dieG, _dieB;
3     private float _numbR, _numbG, _numbB;
4
5     public Theme() {
6         // Die color initialized to grey
7         _dieR = .345f;
8         _dieG = .345f;
9         _dieB = .345f;
10        // Number color initialized to white
11        _numbR = 1f;
12        _numbG = 1f;
13        _numbB = 1f;
14    }
15    [...]
16 }
```

Listing 3.7: Theme Class

I file salvati sono quindi uno per ogni tema (es. *theme1.dice*), uno per ogni preset (es. *preset1.dice*) e il file di selezione denominato *default.dice*. Ognuno di questi file viene salvato all'interno della cartella di salvataggio dell'applicazione, dipendente quindi dal dispositivo sulla quale viene installata.

Capitolo 4

Conclusioni e sviluppi futuri

Dato l'utilizzo di Unity l'aggiunta di funzionalità all'applicazione è semplice e richiede poche modifiche al codice preesistente.

Tramite test svolti sui nostri dispositivi (Samsung S20 FE per ARCore, iPhone 11 per ARKit) abbiamo appurato:

- le differenze tra ARCore e ARKit sono molteplici, di conseguenza l'applicazione su iOS risulta decisamente più fluida, precisa nella creazione di piani e nel tracking degli oggetti. ARCore, a differenza di ARKit, è costretta ad utilizzare API e, non potendo accedere a metodi di più basso livello forniti dal dispositivo, ne risente molto a livello di performance.
- lo spawning di un numero molto più elevato di elementi (300+ dadi) non comporta perdita di frame o crash dell'applicazione, tuttavia non possiamo garantire che ciò avvenga anche su dispositivi diversi, soprattutto se provvisti di hardware di minore potenza;

Ovviamente, ci aspettiamo nei prossimi anni predizioni sempre più accurate grazie a miglioramenti sulle intelligenze artificiali impiegate da ARCore ed ARKit e possibili nuove aggiunte da parte di Unity al package AR Foundation per migliorare l'esperienza AR ed ampliare le sue funzionalità.

Riteniamo inoltre che sarebbe interessante se, dato che sempre più dispositivi sia Android che iOS hanno multiple camere, gli sviluppatori di ARCore ed ARKit aggiungessero per i dispositivi che lo consentono di utilizzare previsioni tramite visione stereo anziché mono.

Sviluppi Futuri

Possibili sviluppi futuri al progetto:

- aggiungere la possibilità di rendere invisibili gli oggetti aggiunti alla scena e degli elementi della Table Mode;
- aggiungere in Table Mode la possibilità di proiettare virtualmente sul tavolo dei tabelloni/mappe di gioco e l'utilizzo di pedine virtuali;
- implementare l'utilizzo di ARCore Depth Lab (Android) e l'utilizzo del Lidar sui dispositivi iOS che lo supportano per aumentare la precisione della depth estimation e della object recognition;

Bibliografia

- [1] Apple. *AR Kit*. URL: <https://developer.apple.com/documentation/arkit/>.
- [2] Google. *AR Core*. URL: <https://developers.google.com/ar/develop>.
- [3] Unity Technologies. *AR Foundation*. URL: <https://docs.unity3d.com/Packages/com.unity.xr.arfoundation@4.2/manual/index.html>.
- [4] Unity Technologies. *AR Foundation Trackable Managers*. URL: <https://docs.unity3d.com/Packages/com.unity.xr.arfoundation@3.0/manual/trackable-managers.html>.