UNIVERSITA' DEGLI STUDI DI
NAPOLI FEDERICO II

Scuola Politecnica e delle Scienze di Base
Corso di Laurea in Ingegneria Informatica

Elaborato finale in **Fondamenti di Informatica**

Progettazione di un'applicazione di Augmented Reality con Unity

Anno Accademico 2020/2021

Candidato:

Lorenzo Manna

matr. N46003667

Indice

Introduzione

La realtà aumentata (AR, Augmented Reality) è un'integrazione di tecnologie volta a combinare informazioni dell'ambiente fisico con informazioni aggiuntive; utilizzando varie tecnologie (computer vision, analisi dell'ambiente attraverso sensori, etc) permette l'interazione tra oggetti fisici e virtuali.

La realtà aumentata può essere utilizzata in diversi campi: videogiochi, veicoli e persino operazioni chirurgiche: recentemente è infatti stato eseguito il primo intervento chirurgico con l'ausilio della realtà aumentata.

Può essere implementata grazie a dispositivi semplici che attualmente sono di utilizzo quotidiano, come cellulari, tablet e webcam: inquadrando un qualsiasi oggetto, grazie alla realtà aumentata (AR) potremo vedere informazioni che verranno sovrapposte a quelle reali.

Gli utilizzi sono molteplici e vanno aumentando:

- Assistenza per gli ipovedenti: inquadrando un documento, se ne permette la lettura utilizzando la sintesi vocale.
- In campo medico: in chirurgia, come ausilio nella formazione e nel tirocinio dei nuovi medici e anche come supporto nella diagnosi di alcuni tipi di patologie.
- In ambito turistico: inquadrando un monumento, per esempio, sarà possibile ricavarne tutte le informazioni aggiuntive desiderate.
- Nei veicoli: HUD (Heads Up Display) fornisce informazioni sullo stato del veicolo permettendo al guidatore di non distogliere lo sguardo dal percorso.

1 Strumenti

Allo scopo di facilitare il porting, si utilizzano framework che, attraverso forti astrazioni, permettono di ridurre al minimo la dipendenza dell'applicazione dalla piattaforma di destinazione.

Per facilitare lo sviluppo multipiattaforma ho utilizzato Unity[1] come engine e Vuforia[2] come framework AR.

Per lo sviluppo sono stati utilizzati vari strumenti per facilitare la gestione del codice e la progettazione, tra cui Git[3] e PlantUML[4].

1.1 Unity

Unity è uno dei game engine più diffusi sul mercato. Permette lo sviluppo di giochi multipiattaforma 2D e 3D senza troppa difficoltà. Utilizza un'architettura Entity-Component, dove ogni "entità" (GameObject) contiene un numero di script (Component) che ne definisce il comportamento[5]; alcuni sono forniti da Unity, ma è sempre possibile crearne nuovi per implementare funzionalità specifiche, utilizzando lo scripting in C#[6]. Inoltre i GameObject possono essere combinati per formare alberi di oggetti per definire composizioni complesse. Il motore ha un editor che permette di posizionare gli oggetti in una scena e impostare le variabili degli script consentendo di visualizzare lo stato della scena, facilitandone così la costruzione.

1.2 Vuforia

Vuforia è una SDK che permette di implementare con facilità funzionalità per la realtà aumentata in applicazioni, in particolare utile per aggiungere funzionalità di realtà aumentata in Unity. Offre funzionalità di riconoscimento e tracking di diversi tipi di oggetti, chiamati target, che comprendono immagini, oggetti 3D di diverse forme, anche complesse, e ambienti permettendo di implementarle nella propria applicazione[7].

Oltre a offrire le funzionalità offline, rende disponibile anche un servizio cloud per alleggerire il dispositivo di parte del lavoro necessario al riconoscimento, permettendo di svolgere compiti di tracking più complessi su dispositivi più deboli.

Vuforia necessita di un riferimento per riconoscere ogni oggetto da tracciare, costruendo un database di target. I riferimenti possono essere ottenuti dall'applicazione in due modi: attraverso un database in rete conservato sul servizio cloud di Vuforia, che permette di aggiornare in remoto i target per tutti gli utenti, oppure generandolo localmente a runtime, così da permettere di scegliere sul momento i target.

1.3 Git

Git è un version control system, ovvero software per la gestione, archiviazione e commento delle modifiche. Consente di tenere aggiornato il codice su più dispositivi, tra più sviluppatori, di tornare a versioni precedenti in caso di regressioni, di creare branch per introdurre modifiche senza rischiare di danneggiare il vecchio codice.

I progetti sono organizzati in repository (repo) e le modifiche sono raggruppate in commit che sono ordinati cronologicamente e commentati per permetterne il facile riconoscimento. Permette di salvare l'intera storia del progetto utilizzando poche risorse del disco poiché ogni commit contiene solo le differenze dei file rispetto al precedente.

1.4 GitHub

Servizio di hosting di repo git che offre molti strumenti gratuitamente o a prezzi accessibili, in base alle funzionalità richieste[8]. Permette di creare gratuitamente repo private anche di grandi dimensioni. Offre un servizio di continuous integration (CI) per

implementare testing automatizzato attraverso GitHub Actions. Inoltre incluso nel servizio di base vi è anche un semplice servizio di hosting web per pubblicare il sito del proprio progetto[9].

1.5 PlantUML

PlantUML è un generatore open source di diagrammi per la progettazione e documentazione del software che utilizza un linguaggio di scripting per disegnare i diagrammi. Essendo basato su testo, è facilmente utilizzabile con git permettendo di utilizzare tutte le funzionalità di versioning tradizionali, il che consente di tenere traccia delle modifiche apportate, aiuta a non perdere dati e a collaborare.

2 Progettazione

2.1 Obiettivo del lavoro

Lo scopo di questo elaborato è di progettare e realizzare un prototipo di un'applicazione di realtà aumentata per smartphone Android.

L'applicazione in questione è un gioco di caccia al tesoro in cui il giocatore deve, sulla base degli indizi ottenuti, trovare ed inquadrare *tesori*, immagini che sono state scelte dall'autore del gioco. Al ritrovamento di un tesoro, il giocatore deve completare una breve sfida per avanzare ed ottenere il suggerimento successivo.

La caccia al tesoro può essere ottenuta in due modi: o è locale, caricata insieme all'applicazione, oppure è scaricata da un server e quindi può essere aggiornata più facilmente.

2.2 Struttura generale

Il gioco è composto da due scene principali:

- *TitleScreen*: permette di avviare la partita.
- *TreasureHunt*: la caccia al tesoro vera e propria.

All'avvio della partita vengono caricati i dati della caccia al tesoro, inizializzando i target di Vuforia e successivamente si passa alla scena *TreasureHunt*.

Durante la caccia al tesoro viene mostrato al giocatore il video dalla videocamera dello smartphone.

Quando viene inquadrato un *target* (uno dei tesori) se si tratta di quello giusto, ossia se è trovato nella sequenza corretta, viene avviato un minigioco al completamento del quale viene mostrato all'utente il suggerimento per trovare il prossimo tesoro e vengono aggiunti punti al punteggio totale.

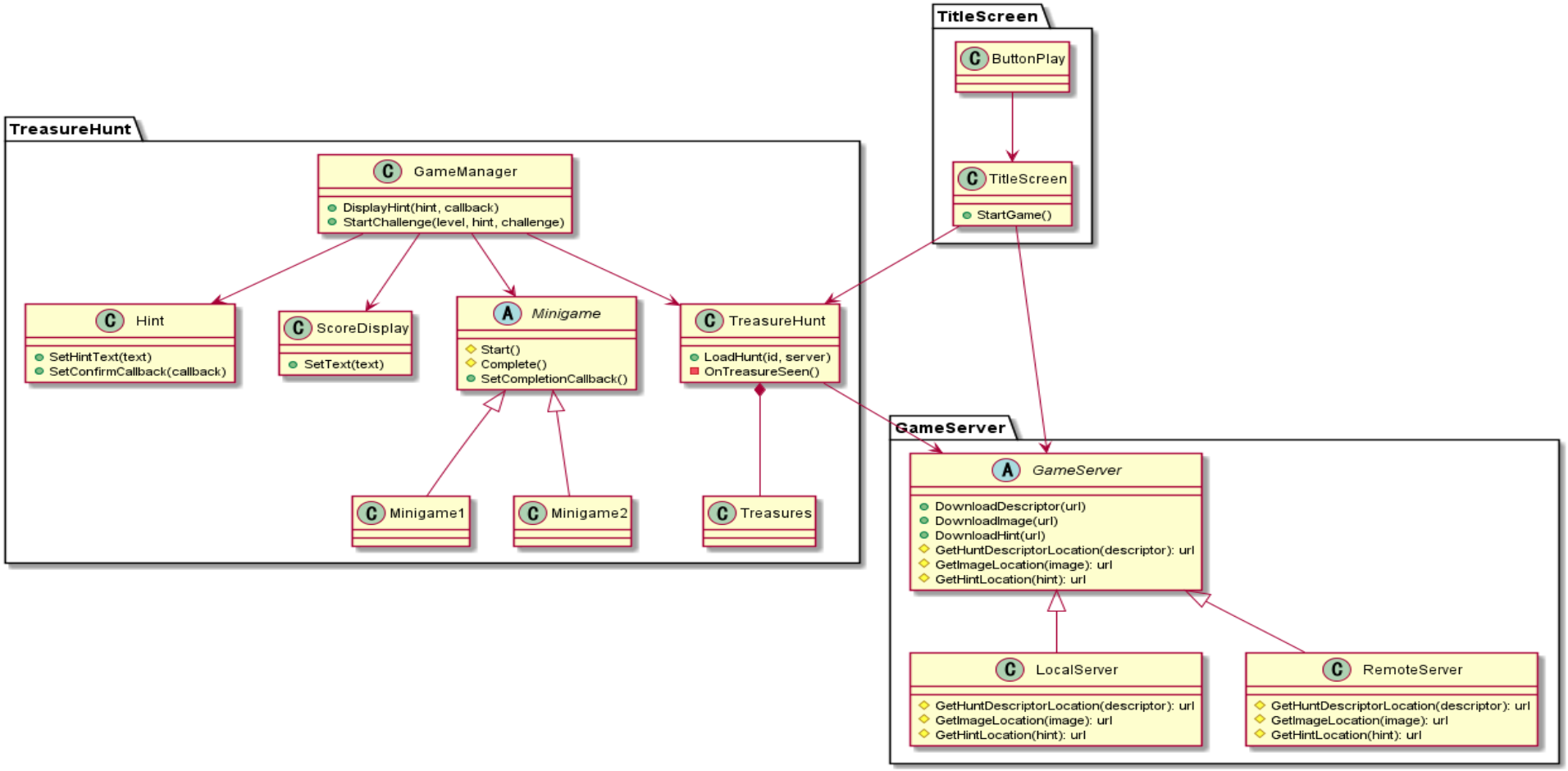
2.3 Architettura

Quando viene premuto il tasto di avvio della partita il *TitleScreen* crea un oggetto di classe base *GameServer* che, se la partita è locale, è *LocalServer*, mentre se è scaricata da remoto è *RemoteServer*. Questo oggetto contiene i metodi per scaricare le varie risorse che vengono utilizzate per costruire l'oggetto *TreasureHunt*, che rappresenta l'intera caccia al tesoro, costruendo e conservando i target di Vuforia e i suggerimenti associati.

Viene caricata quindi la scena *TreasureHunt* che contiene un oggetto *GameManager* che tiene traccia del progresso nel gioco e viene informato di ogni rilevamento di un tesoro da parte dell'oggetto *TreasureHunt*, attraverso la funzione *OnTreasureSeen()*.

Il GameManager mantiene il punteggio totale e tiene traccia dei tesori trovati attraverso un contatore che viene incrementato a ogni ritrovamento. La funzione *StartChallenge()*, che viene chiamata quando dovrebbe esserci una sfida, controlla questo contatore e se il tesoro rilevato è nell'ordine giusto, avvia il minigioco e al completamento di esso aggiunge i punti al punteggio, incrementa il contatore e mostra il suggerimento per il tesoro successivo.

Il suggerimento è un insieme di oggetti raggruppati sotto il GameObject *Hint* che contiene un campo di testo e un tasto per tornare al gioco. Di default il suggerimento è nascosto, ma sempre presente nella scena. Quando il minigioco è completato, viene attivato e diventa quindi visibile.


TitleScreen
ButtonPlay
TitleScreen
StartGame()
TreasureHunt
GameManager
DisplayHint(hint, callback)
StartChallenge(level, hint, challenge)
Hint
SetHintText(text)
SetConfirmCallback(callback)
ScoreDisplay
SetText(text)
Minigame
Start()
Complete()
SetCompletionCallback()
TreasureHunt
LoadHunt(id, server)
OnTreasureSeen()
Minigame1
Minigame2
Treasures
GameServer
GameServer
DownloadDescriptor(url)
DownloadImage(url)
DownloadHint(url)
GetHuntDescriptorLocation(descriptor): url
GetImageLocation(image): url
GetHintLocation(hint): url
LocalServer
GetHuntDescriptorLocation(descriptor): url
GetImageLocation(image): url
GetHintLocation(hint): url
RemoteServer
GetHuntDescriptorLocation(descriptor): url
GetImageLocation(image): url
GetHintLocation(hint): url

2.3.1 GameServer

Classe astratta che rappresenta un server del gioco, ovvero il fornitore dei dati della caccia al tesoro.

Implementa **coroutine** (funzioni asincrone) per ottenere i dati:

- *DownloadDescriptor*
- *DownloadImage*
- *DownloadHint*

e dei **template method**:

- *GetHuntDescriptorLocation*
- *GetImageLocation*
- *GetHintLocation*

che vengono implementati dalle sottoclassi *LocalServer* e *RemoteServer* per definire la localizzazione dei dati che possono provenire dal disco locale o da rete.

Le coroutine sono un metodo leggero per implementare operazioni asincrone senza aver bisogno di multithreading. Sono un sistema di multitasking cooperativo[10], ovvero ogni coroutine si interrompe volontariamente con uno *yield* durante l'esecuzione in corrispondenza di operazioni lente, in modo da permettere l'esecuzione delle altre.

Sono state scelte le coroutine perché non essendo bloccanti, permettono alla logica del gioco di continuare a eseguire mentre vengono svolte operazioni lente, come la lettura dei file o il download da rete.

Il pattern del template method è stato scelto poiché permette di aggiungere facilmente nuove funzionalità attraverso l'estensione della classe base.

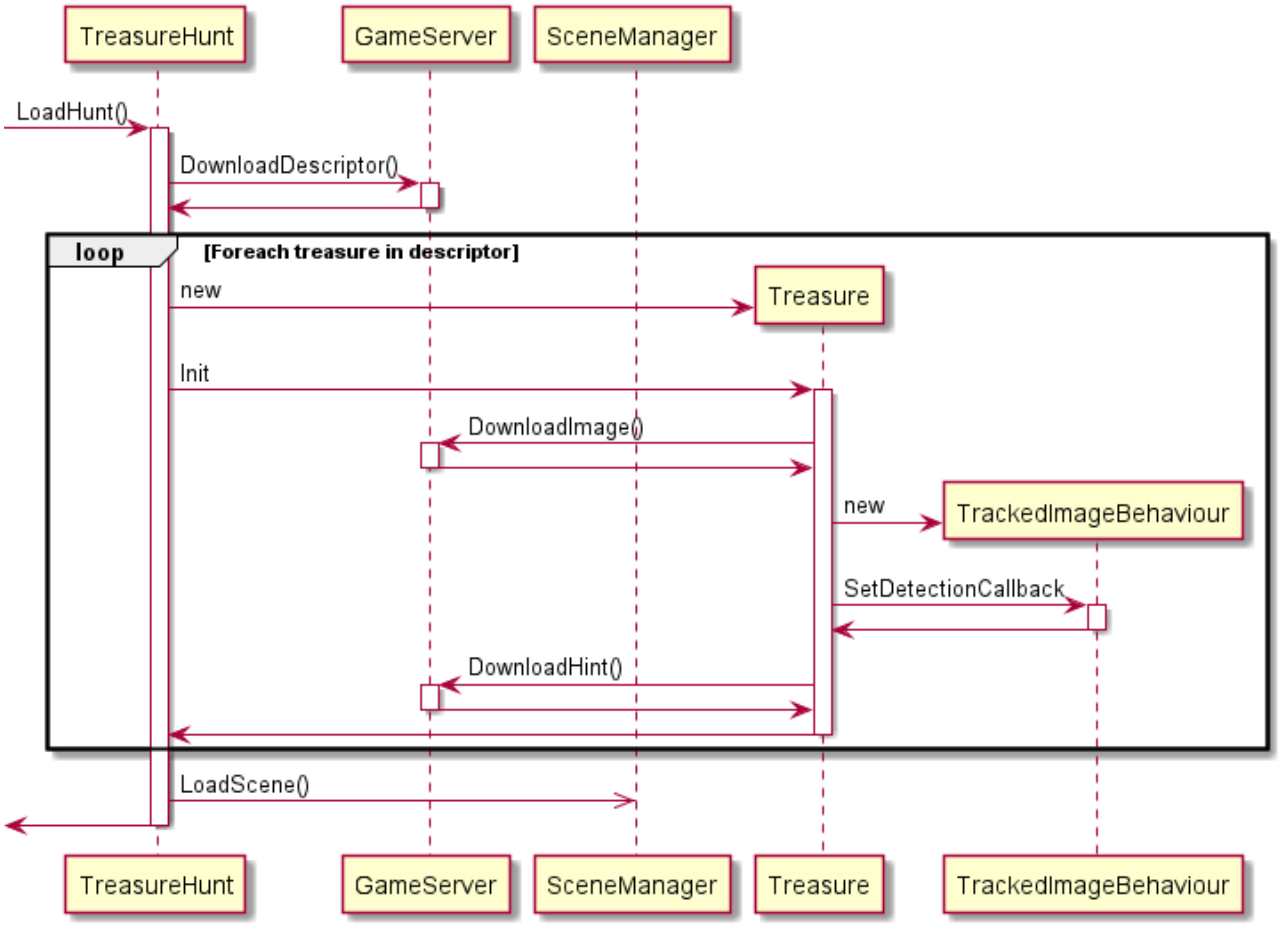
2.3.2 TreasureHunt

Classe che rappresenta la caccia al tesoro.

Implementa le funzioni:

- *LoadHunt*
- *OnTreasureSeen*

La funzione *LoadHunt* accetta l'id della caccia al tesoro e il server e utilizzando coroutine scarica i dati attraverso il *GameServer* e costruisce i *Treasure* che vengono utilizzati per il tracking.



TreasureHunt chiama i vari metodi del *GameServer* per ottenere le risorse, passando l'id della risorsa e un callback da eseguire al completamento dell'operazione. Il callback permette di impostare i valori necessari dai dati ottenuti senza dover attendere bloccando l'esecuzione, migliorando quindi la responsività e riducendo i tempi di caricamento.

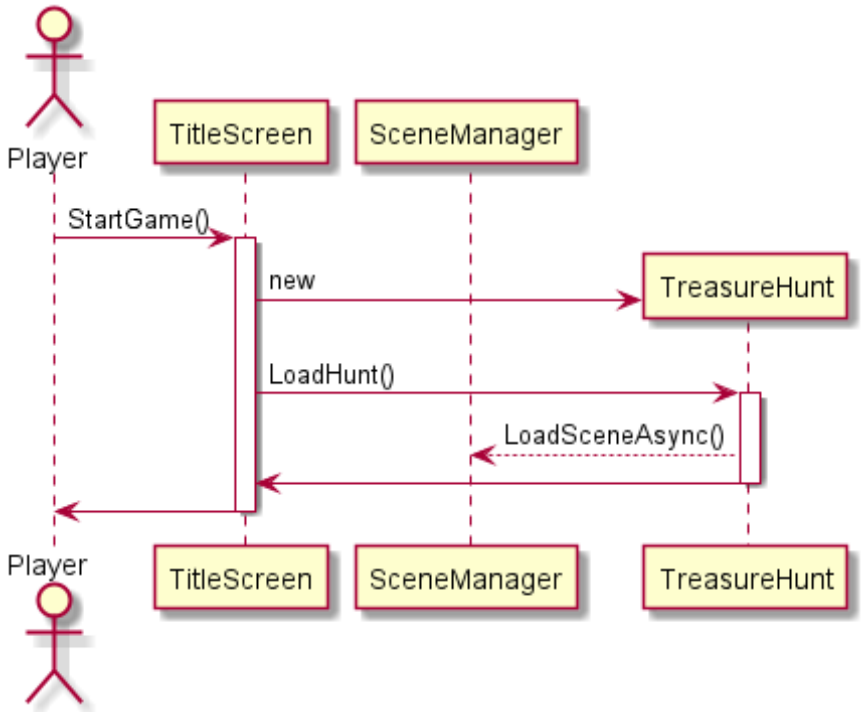
OnTreasureSeen viene passata ai *Treasure* come callback per ricevere la notifica del ritrovamento del tesoro. Questo metodo viene utilizzato dalla *TreasureHunt* per notificare

al *GameManager* il ritrovamento di un tesoro, passando solo i dati necessari come l'id del tesoro, il suggerimento associato e il minigioco da avviare.

2.3.3 TitleScreen

Classe che rappresenta l'interfaccia del menu principale.

Implementa le operazioni di avvio del gioco. Chiamando *StartGame*, vengono creati un *GameServer* e una *TreasureHunt*. La *TreasureHunt* viene inizializzata utilizzando il *GameServer*.



2.3.4 GameManager

Classe che rappresenta una partita in corso.

Implementa le funzioni:

- *DisplayHint*
- *StartChallenge*

DisplayHint mostra al giocatore il testo del suggerimento e permette di definire un callback che viene eseguito alla conferma di lettura da parte dell'utente.

StartChallenge avvia il minigioco indicato se il livello attuale corrisponde a quello

specificato. Al completamento del minigioco, incrementa il contatore del progresso, assegna i punti prestabiliti e mostra il suggerimento.

2.3.5 ScoreText

Classe che rappresenta il campo di testo per mostrare il punteggio attuale.

2.3.6 Hint

Classe che rappresenta il campo di testo per mostrare il suggerimento e un tasto per la conferma.

Implementa le funzioni:

- *SetHintText*
- *SetConfirmCallback*
- *Confirm*

SetHintText permette di impostare il testo da mostrare, modificando l'attributo text dell'oggetto Text figlio.

SetConfirmCallback permette di impostare il callback chiamato alla pressione del tasto di conferma.

Confirm viene chiamata alla pressione del tasto e disattiva l'oggetto in modo da nasconderlo e chiama il callback, se impostato.

2.3.7 Minigame

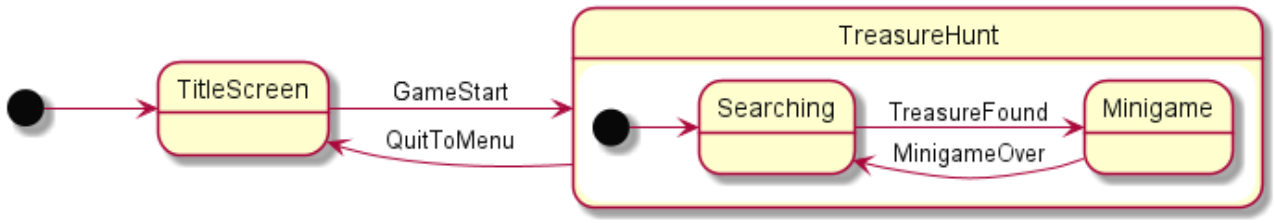
Classe astratta che rappresenta un generico minigioco i cui metodi sono template implementabili dalle classi derivate, ovvero i minigiochi concreti:

- *Start*
- *SetCompletionCallback*

Servono rispettivamente a inizializzare il minigioco e a registrare il callback di notifica del completamento.

2.4 Realizzazione

Il gioco è diviso in due scene: TitleScreen e TreasureHunt.



2.4.1 TitleScreen

La scena TitleScreen rappresenta la schermata di avvio del gioco; contiene i tasti per le interazioni e lo script TitleScreen.cs che definisce le funzionalità.

Il tasto principale è StartGame la cui pressione esegue l'omonima funzione definita in TitleScreen.cs, che avvia la partita caricando i dati necessari.

La funzione StartGame comincia creando il server appropriato, in questo caso un LocalServer, e successivamente crea un oggetto TreasureHunt che servirà per conservare i dati della partita.

Per inizializzare i dati, viene chiamata la funzione LoadHunt. Questo è necessario, poiché TreasureHunt è un MonoBehaviour, cioè un componente di Unity, per cui non è possibile chiamare esplicitamente il costruttore, come si può per normali classi. La funzione LoadHunt usa il server creato in precedenza per scaricare i dati associati alla caccia al tesoro specificata dall'id per creare gli oggetti per il tracking. Il server fornisce tre coroutine per scaricare i dati, una per ogni tipologia: DownloadDescriptor, DownloadImage e DownloadHint.

Queste funzioni hanno due parametri, l'id della risorsa da ottenere e un callback che viene chiamato al completamento dell'operazione così da permettere al chiamante di utilizzare i risultati senza aver bisogno di attendere e potendo così procedere a eseguire le successive coroutine.

La funzione DownloadDescriptor permette di scaricare il descrittore della caccia al tesoro,

ovvero un json che contiene tutti i dati per costruire l'oggetto:

- *format*: stringa che indica la versione del formato del descrittore.
- *hint*: stringa che indica l'url del suggerimento che porta al primo tesoro.
- *treasures*: array di oggetti che definiscono i tesori:
  - *type*: stringa che indica il tipo di tesoro (al momento solo "image")
  - *url*: stringa che indica l'url della risorsa che rappresenta il tesoro.
  - *hint*: stringa che indica l'url del suggerimento ottenuto da questo tesoro.
  - *minigame*: stringa che indica il tipo di minigioco da eseguire (non implementato)

Questa struttura è stata scelta per semplificare l'implementazione di un server, poiché permette di costruirne uno con una semplice API REST separando la parte statica (immagini e testo dei suggerimenti) dalla parte dinamica del descrittore, che potrebbe essere modificato rapidamente, se necessario.

DownloadImage e DownloadHint permettono di scaricare la rispettiva risorsa.

Per unificare il codice per il caricamento da disco locale e da rete, ho utilizzato i template method GetHuntDescriptorLocation, GetImageLocation e GetHintLocation, che restituiscono gli URL delle risorse, poiché esse possono essere ottenute utilizzando UnityWebRequest che permette di costruire richieste HTTP che possono prelevare dati locali e remoti.

Nel caso del LocalServer, gli URL punteranno a risorse sul disco locale, contenute nella cartella di Unity StreamingAssets.

Nel caso del RemoteServer, gli URL punteranno a risorse sul server e quindi saranno utilizzati per eseguire richieste HTTP.

Alla fine della funzione LoadHunt, viene caricata la scena TreasureHunt e quindi si passa ad essa.

Per implementare il tracking, sono utilizzate le funzionalità fornite da Vuforia.

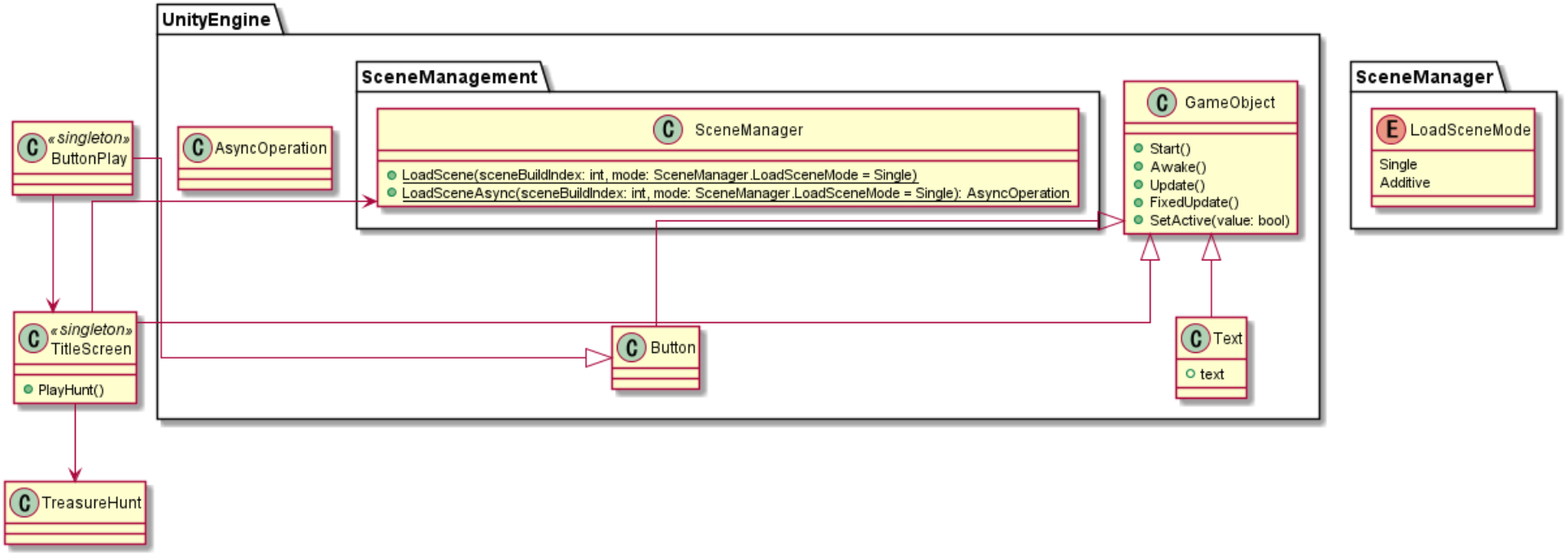
Vuforia permette di registrare un **observer** per ricevere notifiche sugli eventi, in questo caso è stata estesa la classe DefaultObserverEventHandler con la classe TrackedImageBehaviour che implementa le funzionalità necessarie.

Sovrascrivendo il metodo HandleTargetStatusChanged la classe TrackedImageBehaviour notifica attraverso un callback registrato alla creazione l'inizio del tracciamento dell'oggetto.

Segue l'implementazione del GameServer e l'implementazione del LocalServer come esempio.

```
public abstract class GameServer {
    public delegate void DownloadCallback<T>(T result);

    protected abstract string GetHuntDescriptorLocation(string id);
    protected abstract string GetImageLocation(string url);
    protected abstract string GetHintLocation(string url);

    public virtual IEnumerator DownloadDescriptor(string id,
        DownloadCallback<TreasureHunt.HuntDescriptor> callback) {
        using (var request = UnityWebRequest.Get(GetHuntDescriptorLocation(id))) {
            yield return request.SendWebRequest();

            switch (request.result) {
                case UnityWebRequest.Result.Success:
                    var huntDescriptor = JsonUtility
                        .FromJson<TreasureHunt.HuntDescriptor>(
                            request.downloadHandler.text
                        );
                    callback(huntDescriptor);
                    break;
                default:
                    Debug.LogError(request.error);
                    break;
            }
        }
    }

    public virtual IEnumerator DownloadImage(string url,
        DownloadCallback<Texture2D> callback) {
        using (var request = UnityWebRequestTexture.GetTexture(GetImageLocation(url))) {
            yield return request.SendWebRequest();

            switch (request.result) {
                case UnityWebRequest.Result.Success:
                    callback(DownloadHandlerTexture.GetContent(request));
                    break;
                default:
                    Debug.LogError(request.error);
                    break;
            }
        }
    }

    public virtual IEnumerator DownloadHint(string url,
        DownloadCallback<string> callback) {
        using (var request = UnityWebRequest.Get(GetHintLocation(url))) {
            yield return request.SendWebRequest();

            switch (request.result) {
                case UnityWebRequest.Result.Success:
                    var hint = request.downloadHandler.text;
                    callback(hint);
                    break;
                default:
                    Debug.LogError(request.error);
                    break;
            }
        }
    }
}
```

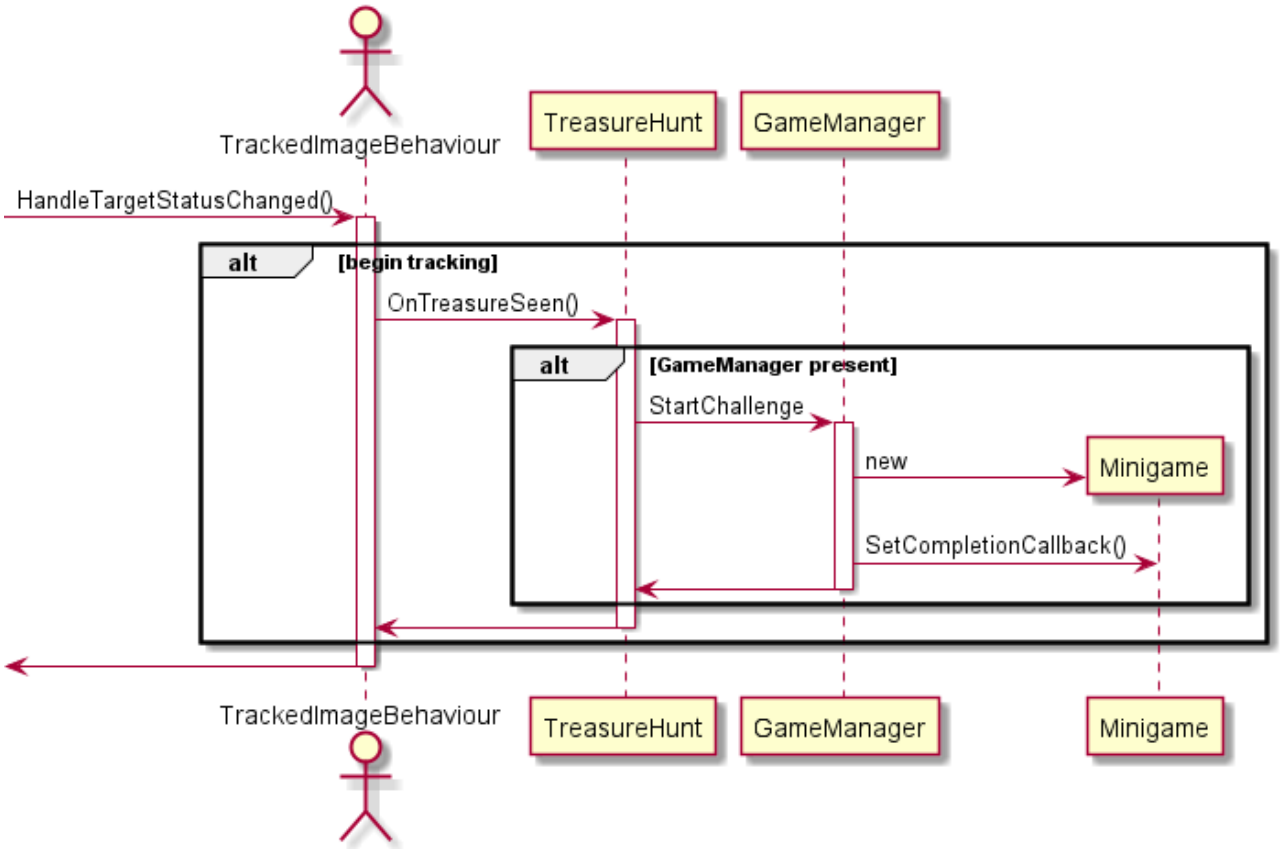
```
public class LocalServer : GameServer {
    protected override string GetHintLocation(string url) {
        return Path.Combine(Application.streamingAssetsPath, url);
    }

    protected override string GetHuntDescriptorLocation(string id) {
        return Path.Combine(Application.streamingAssetsPath, "hunts", id, "descriptor");
    }

    protected override string GetImageLocation(string url) {
        return Path.Combine(Application.streamingAssetsPath, url);
    }
}
```


UnityEngine
SceneManagement
SceneManager
«singleton»
ButtonPlay
AsyncOperation
SceneManager
LoadScene(sceneBuildIndex: int, mode: SceneManager.LoadSceneMode = Single)
LoadSceneAsync(sceneBuildIndex: int, mode: SceneManager.LoadSceneMode = Single): AsyncOperation
GameObject
Start()
Awake()
Update()
FixedUpdate()
SetActive(value: bool)
LoadSceneMode
Single
Additive
«singleton»
TitleScreen
PlayHunt()
Button
Text
text
TreasureHunt

2.4.2 TreasureHunt

La scena è composta da vari GameObject. Il principale è il GameManager, che rappresenta la partita attuale, tiene traccia dei tesori trovati e gestisce i minigiochi e il display dei suggerimenti. Permette, attraverso l'inspector dell'editor di Unity, di aggiungere ulteriori minigiochi semplicemente inserendo il prefab (insieme di GameObject serializzati che possono quindi essere caricati quando desiderato) che li rappresenta. Per rendere più accessibile la configurazione ho scelto di mostrare nell'inspector una lista di minigiochi che viene poi utilizzata per costruire un dizionario che associa i nomi dei minigiochi, ottenuti dal nome del prefab, al GameObject stesso per rendere più rapidi gli accessi successivi.

L'oggetto TreasureHunt viene conservato dalla scena precedente e, al ritrovamento di un tesoro, attraverso la funzione OnTreasureSeen, comunica l'evento al GameManager avviando il minigioco.

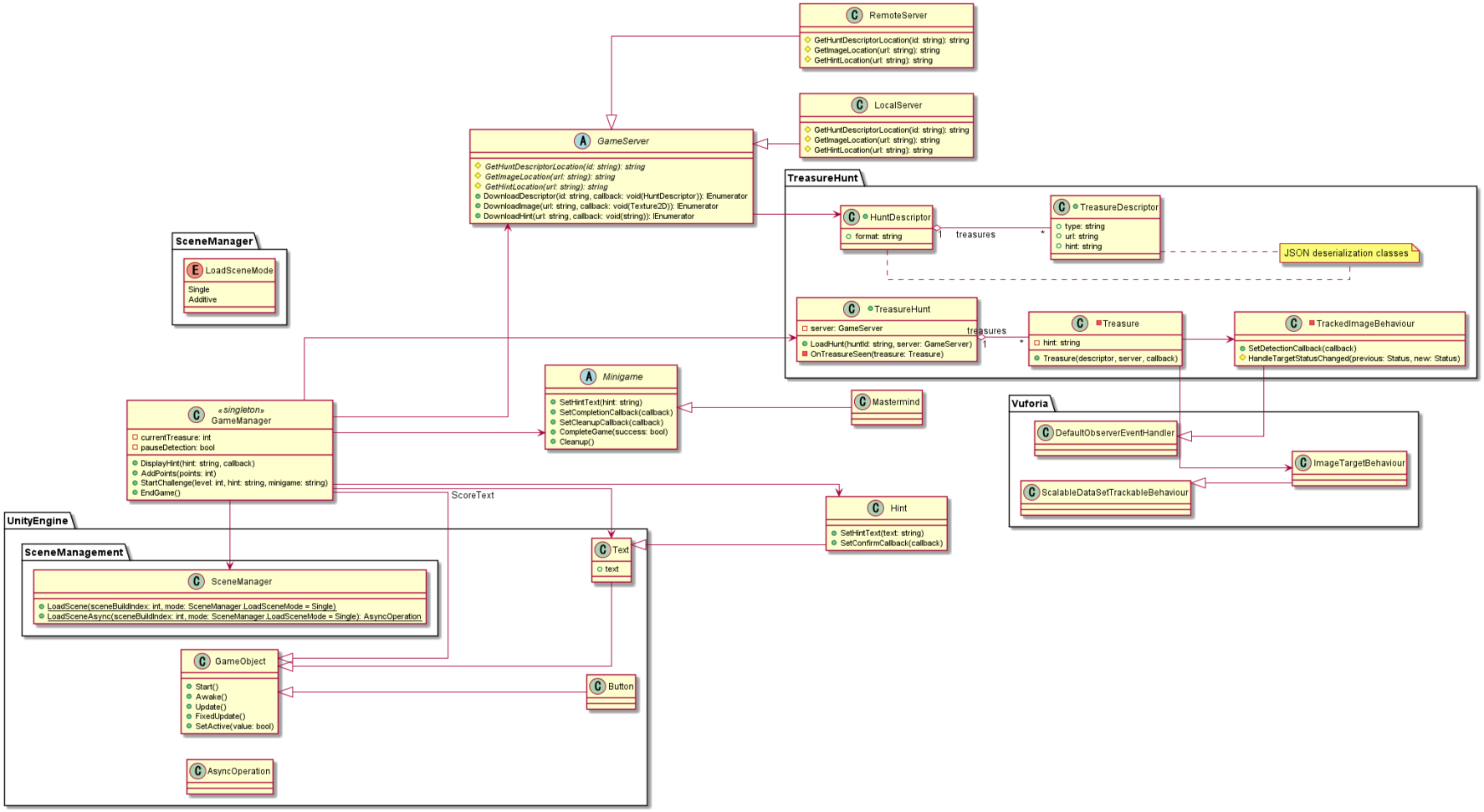


L’oggetto ScoreText viene aggiornato ogni volta che viene modificato il punteggio.

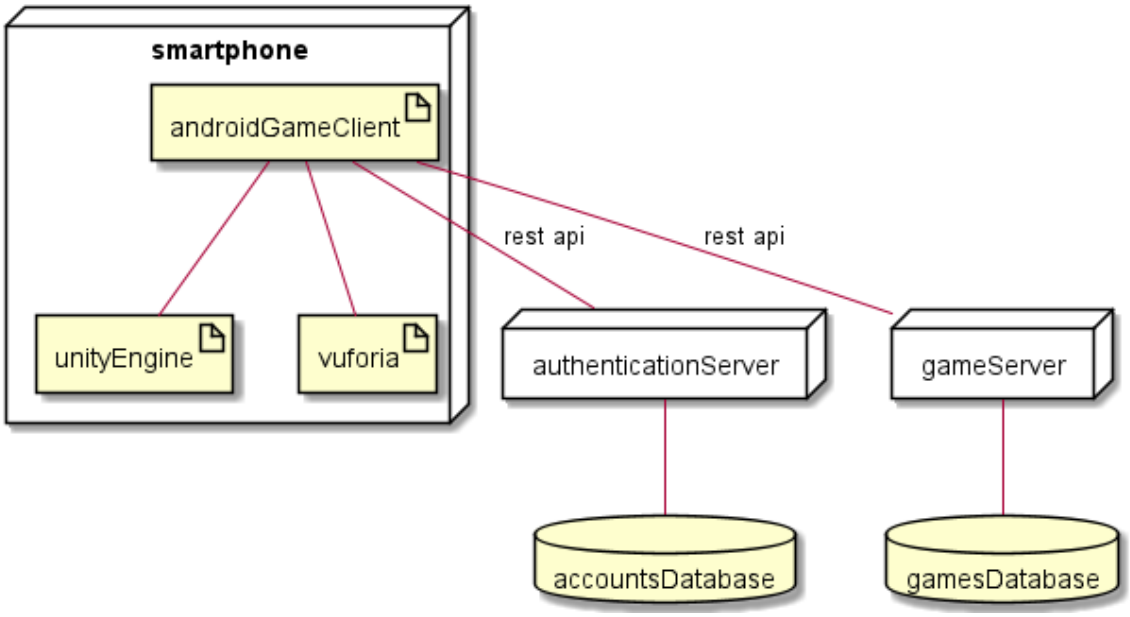
L’oggetto Hint viene attivato quando è necessario mostrare un suggerimento. È il parent di un Text e di un Button che sono rispettivamente il campo di testo per il suggerimento e il tasto di conferma. Hint possiede un omonimo script contenente funzioni per facilitare l’utilizzo di questi due oggetti.

Attraverso la funzione StartChallenge, il GameManager crea un oggetto Minigame che è una composizione di più oggetti. Poiché Minigame è un MonoBehaviour, non è possibile utilizzare l’ereditarietà per definire comportamenti alternativi, quindi è emulato il pattern dei template method attraverso composizione e callback: alla radice del minigioco si ha un oggetto contenente i componenti Minigame e un componente che definisce la logica del minigioco specifico. Quest’ultimo, all’avvio, registra le sue funzioni come callback in Minigame, così da poter utilizzare la stessa interfaccia.

```
public class GameManager : MonoBehaviour {
    public GameObject[] MinigamesList;
    public GameObject HintPrompt;
    public TextMeshProUGUI ScoreText;

    private Dictionary<string, GameObject> minigames;
    private int currentTreasure = 0;
    private int score = 0;
    private bool pauseDetection = false;

    void Start() {
        VuforiaBehaviour.Instance.VideoBackground.StartVideoBackgroundRendering();
        minigames = new Dictionary<string, GameObject>();

        foreach (var minigame in MinigamesList) {
            minigames.Add(minigame.name.ToLower(), minigame);
        }
    }

    public void DisplayHint(string hint,
       HintPrompt.ConfirmCallback confirmCallback = null) {
        pauseDetection = true;
        HintPrompt.GetComponent<HintPrompt>().SetHintText(hint);
        HintPrompt.GetComponent<HintPrompt>().SetConfirmCallback(
             () => { pauseDetection = false; } + confirmCallback
        );
        HintPrompt.SetActive(true);
    }

    private void AddPoints(int points) {
        score += points;
        if (ScoreText != null) ScoreText.text = (score * 15).ToString();
    }

    public void StartChallenge(int level, string hint, string minigame) {
        if (pauseDetection || level != currentTreasure) return;
        pauseDetection = true;
        var game = Instantiate(minigames[minigame.ToLower()]).GetComponent<Minigame>();
        game.SetCompletionCallback((bool status) => {
            Debug.Log("Completed game " + level + " " + status);
            currentTreasure++;
            AddPoints(1);
            DisplayHint(hint, () => {
                game.Cleanup();
            });
        });
    }

    public void EndGame(int level) {
        if (pauseDetection || level != currentTreasure) return;
        var load = SceneManager.LoadSceneAsync("TitleScreen");
        load.allowSceneActivation = false;
        DisplayHint("You win!", () => {
            load.allowSceneActivation = true;
        }) ;
    }
}
```


RemoteServer
GetHuntDescriptorLocation(id: string): string
GetImageLocation(url: string): string
GetHintLocation(url: string): string
LocalServer
GetHuntDescriptorLocation(id: string): string
GetImageLocation(url: string): string
GetHintLocation(url: string): string
GameServer
GetHuntDescriptorLocation(id: string): string
GetImageLocation(url: string): string
GetHintLocation(url: string): string
DownloadDescriptor(id: string, callback: void(HuntDescriptor)): IEnumerator
DownloadImage(url: string, callback: void(Texture2D)): IEnumerator
DownloadHint(url: string, callback: void(string)): IEnumerator
TreasureHunt
HuntDescriptor
format: string
1
treasures
*
TreasureDescriptor
type: string
url: string
hint: string
JSON deserialization classes
TreasureHunt
server: GameServer
LoadHunt(huntId: string, server: GameServer)
OnTreasureSeen(treasure: Treasure)
treasures
1
*
Treasure
hint: string
Treasure(descriptor, server, callback)
TrackedImageBehaviour
SetDetectionCallback(callback)
HandleTargetStatusChanged(previous: Status, new: Status)
SceneManager
LoadSceneMode
Single
Additive
«singleton»
GameManager
currentTreasure: int
pauseDetection: bool
DisplayHint(hint: string, callback)
AddPoints(points: int)
StartChallenge(level: int, hint: string, minigame: string)
EndGame()
Minigame
SetHintText(hint: string)
SetCompletionCallback(callback)
SetCleanupCallback(callback)
CompleteGame(success: bool)
Cleanup()
Mastermind
Vuforia
DefaultObserverEventHandler
ImageTargetBehaviour
ScalableDataSetTrackableBehaviour
ScoreText
Hint
SetHintText(text: string)
SetConfirmCallback(callback)
UnityEngine
SceneManagement
SceneManager
LoadScene(sceneBuildIndex: int, mode: SceneManager.LoadSceneMode = Single)
LoadSceneAsync(sceneBuildIndex: int, mode: SceneManager.LoadSceneMode = Single): AsyncOperation
Text
text
GameObject
Start()
Awake()
Update()
FixedUpdate()
SetActive(value: bool)
Button
AsyncOperation

2.5 Deployment



Attraverso il *RemoteServer* è possibile collegarsi a un server che fornisce le "cacce al tesoro" permettendo di poter aggiungere dinamicamente nuove mappe e quindi nuove esperienze per il giocatore, funzionalità predisposta nell'applicazione ma al momento non implementata.

3 Modularità del progetto

Uno dei requisiti principali che hanno guidato le mie scelte nel corso della progettazione è stato la modularità.

Ho considerato il progetto tenendo conto della modularità sia dal punto di vista della progettazione e manutenzione che dal punto di vista dell'utilizzatore.

In primo luogo il progetto in sé cerca di essere modulare permettendo di definire nuove funzionalità, come ad esempio nuovi server e nuovi minigiochi che possono essere aggiunti semplicemente estendendo le rispettive classi base implementando le nuove funzionalità.

Per gli utilizzatori, invece, è possibile importare facilmente nuove "cacce al tesoro" in modo da variare e personalizzare il gioco. Le "cacce al tesoro" sono interamente descritte da file json che vengono scaricati quando necessario da un server: questo permette di configurare un server che offre i descrittori delle "cacce al tesoro", le immagini che vengono utilizzate per il tracking e i suggerimenti da mostrare.

4 Dal punto di vista del giocatore

Il gioco è una classica caccia al tesoro leggermente modificata. Il gioco classico prevede la ricerca di un singolo tesoro seguendo un percorso di indizi, invece questa variante prevede il ritrovamento di vari piccoli tesori, in un ordine prestabilito, ognuno contenente un suggerimento per il successivo fino ad arrivare a quello finale, accumulando punti ad ogni step.

Per installare il gioco è sufficiente installare l'apk.

Se si desidera, è possibile ricostruire il progetto, hostato su GitHub nella repo github.com/Lory9811/ARTreasureHunt, importandolo in Unity (testato nella versione 2021.1.21f1), inserendo la propria chiave di Vuforia nel file *VuforiaConfiguration.asset* in *Assets/Resources/* secondo l'esempio in *ExampleVuforiaConfiguration.asset* o attraverso la configurazione di Vuforia in Unity.

Per ricevere la propria chiave è necessario creare un account sul sito developer.vuforia.com e creare una nuova applicazione dal License Manager.

Le funzionalità di Vuforia sfruttate sono solo quelle offline, quindi non è necessario un tier più alto di quello gratuito, poiché non verranno utilizzati i database online e le funzionalità di riconoscimento basate sul cloud.

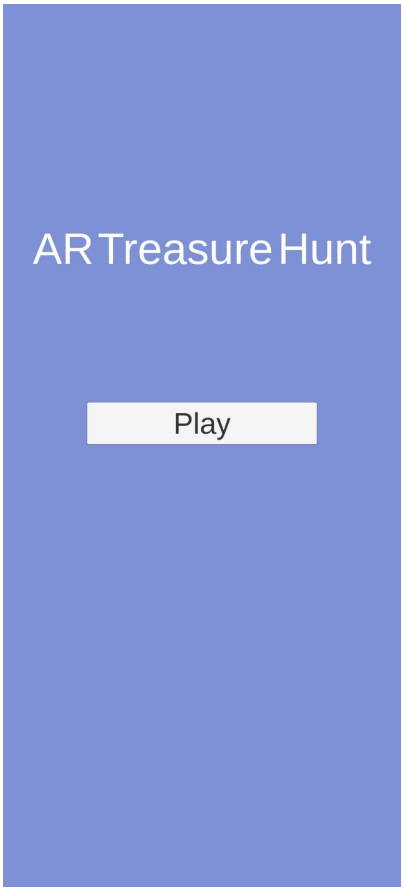
Per personalizzare il gioco è possibile sostituire o aggiungere file in *Assets/StreamingAssets/* per costruire nuove cacce al tesoro con i propri marker e suggerimenti. Se si desidera aggiungere nuove fonti di dati è necessario estendere la classe *GameServer* o modificare una delle esistenti.

Il gioco è stato testato su un OnePlus Nord 2, ma dovrebbe funzionare sulla maggior parte di dispositivi Android partendo dalla versione 7.0, provvisti di videocamera sul retro. Le prestazioni del tracciamento possono variare a seconda della risoluzione della videocamera.

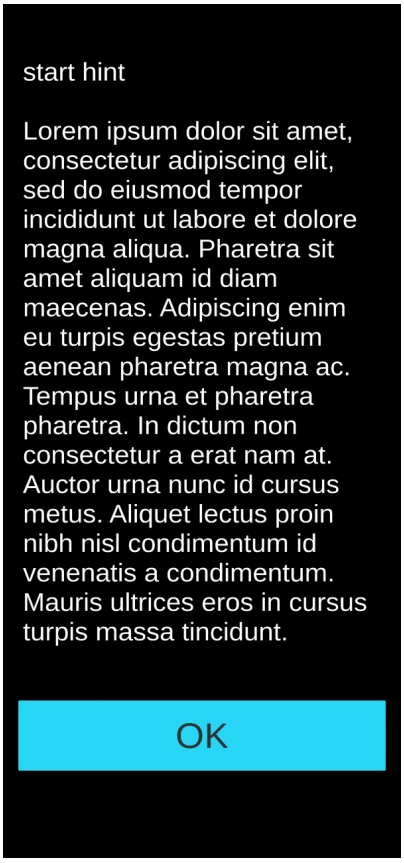
Per cominciare la partita il giocatore riceve un suggerimento che lo indirizza verso il ritrovamento del primo tesoro. Trovato il tesoro deve completare una sfida che consiste in un breve minigioco; se il tesoro trovato non rispetta l'ordine prestabilito, il minigioco non viene avviato ed il giocatore deve trovare quello corretto. La partita si conclude quando tutti i tesori sono stati trovati ed i minigiochi completati.

4.1 Esempio di partita

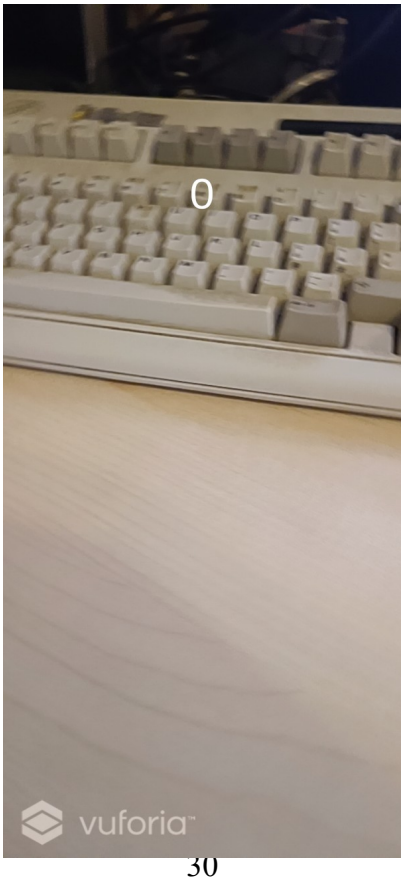
All’avvio del gioco compare la schermata principale che mostra il nome del gioco e il tasto di avvio.



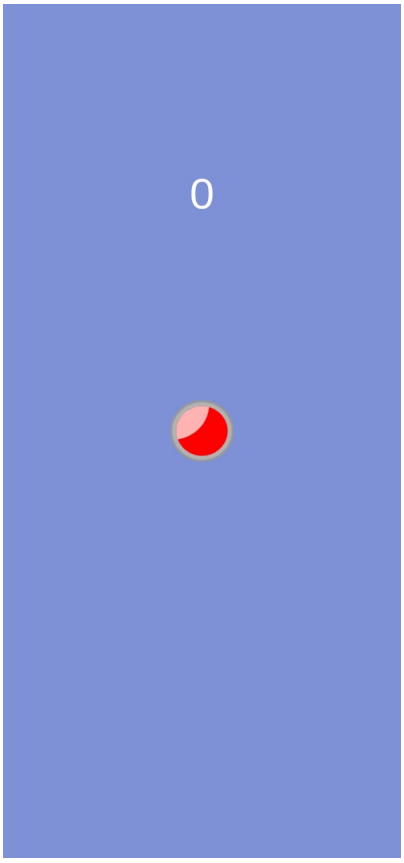
Quando viene avviata la partita, viene mostrato il suggerimento per trovare il primo tesoro



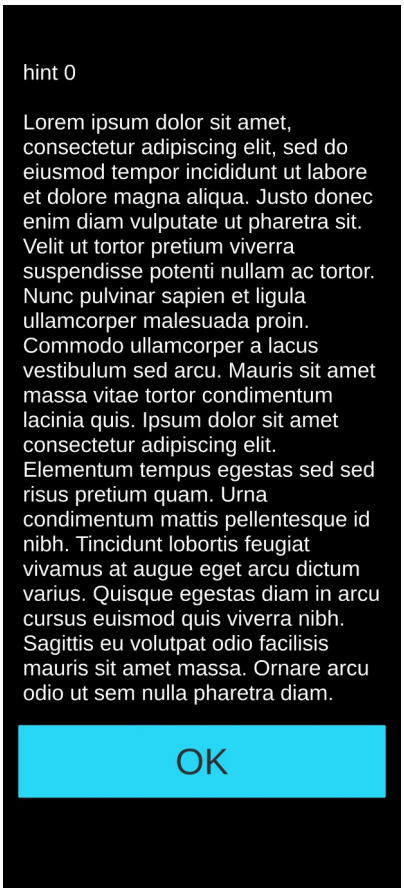
Alla conferma della lettura, si passa alla vista della videocamera del telefono e quindi alla ricerca del tesoro.



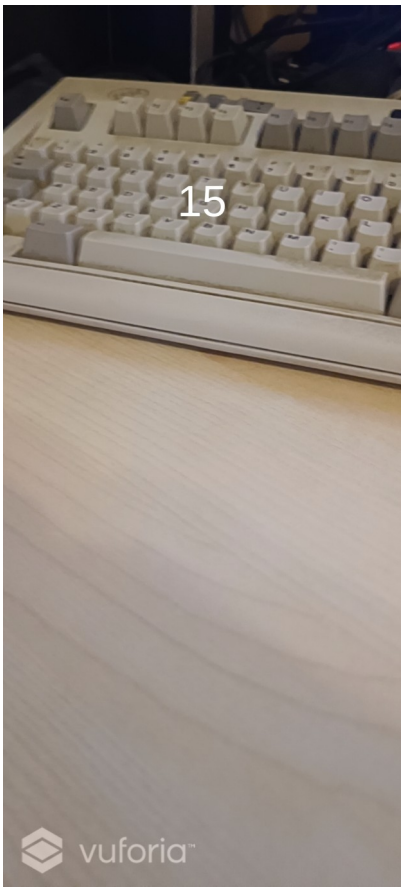
Quando viene inquadrato il tracker, se è nella sequenza giusta, comincia il minigioco.



Completato il minigioco si ottiene il suggerimento successivo



Confermata la lettura si torna alla vista della videocamera e si continua finché non sono stati trovati tutti i tesori.



Conclusioni

L'utilizzo di Vuforia è stato dettato dall'esigenza di fruire di un prodotto che fosse utilizzabile su qualsiasi dispositivo: il framework fa il lavoro necessario in modo soddisfacente e con sufficiente semplicità di utilizzo; purtroppo però questo è limitato da una licenza molto restrittiva nel caso della versione gratuita, che impedisce la pubblicazione dell'applicazione, e da prezzi esagerati che vanno a limitare o addirittura impedire le possibilità di sperimentare con la realtà aumentata. Questo porta a considerare alternative come, ad esempio, altri framework di realtà aumentata come AR Foundation di Unity che però, dipendendo da ARCore, non sono utilizzabili su alcuni dispositivi per mancanza di supporto.

Unity è un motore flessibile, molto adatto allo sviluppo per dispositivi mobili, con una grande community che produce risorse accessibili, ma soffre gravemente a causa di un ecosistema disorganizzato, in particolare negli ultimi anni, in quanto vengono introdotte funzionalità che non vengono poi sfruttate completamente e ne vengono rimosse di fondamentali senza introdurre nuove alternative.
Ovviamente questo porta a prendere in considerazione altri motori come per esempio Godot, motore free e open source per giochi 2D e 3D multipiattaforma, che include anche funzionalità AR.

Riferimenti

[1] Unity, unity.com, 2021/11/04

[2] Vuforia, library.vuforia.com, 2021/11/04

[3] git, git-scm.com, 2021/11/04

[4] PlantUML, plantuml.com, 2021/11/04

[5] Unity Docs, docs.unity3d.com/Manual/GameObjects.html, 2021/11/17

[6] Unity Docs, docs.unity3d.com/Manual/CreatingAndUsingScripts.html, 2021/11/17

[7] Vuforia Docs, library.vuforia.com/getting-started/vuforia-features, 2021/11/17

[8] GitHub, github.com, 2021/11/04

[9] GitHub Pages, pages.github.com, 2021/11/17

[10] Unity Docs, docs.unity3d.com/ScriptReference/Coroutine.html, 2021/11/17