Politecnico di Torino

Laurea Magistrale in Ingegneria del Cinema e dei Mezzi di Comunicazione

Tesi di Laurea Magistrale

Ottimizzazione di Applicazioni in VR: LockedUp

Relatore: Prof. Riccardo Silvio Antonino

Alberto Merletti

ANNO ACCADEMICO 2022-2023

Ringraziamenti

Voglio prima di tutto ringraziare la mia famiglia e i miei amici che mi hanno sempre supportato nel corso di questi due anni. Non dev'essere stato facile ascoltarmi (o almeno far finta di farlo, ma non vi biasimo) mentre gli spiegavo i progetti che stavo facendo nei minimi dettagli, o mentre gli spiegavo perchè il codice che avevo scritto non funzionava, e far finta di essere presi bene tanto quanto me.

Ringrazio poi il mio relatore, il prof Antonino e le collaboratrici/tutor Chiara Mastino e Fabiana Spallone per avermi permesso di lavorare su questo progetto e avermi seguito e aiutato in questi mesi.

Infine voglio ringraziare il Politecnico. Mi sono trovato benissimo in questi due anni, ho imparato tante cose interessanti e conosciuto persone fantastiche, sia tra i miei compagni di studio che tra i prof; mi mancheranno l'ambiente e le persone del Poli e gli aperitivi al Comala dopo lezione.

Indice

Capitolo 1

Introduzione

La realtà virtuale, o VR, è una realtà simulata accessibile tramite un visore che avvolge l'utente. Il lavoro presentato in questa tesi tratta della profilazione e ottimizzazione di applicazioni realizzate in realtà virtuale, occupandosi poi di un caso pratico, ovvero la profilazione e ottimizzazione dell'applicazione LockedUp. LockedUp è un progetto ideato nel 2020 da un gruppo di studenti di Ingegneria del Cinema in collaborazione con il Politecnico di Torino e il National Film Board Of Canada. Attraverso l'applicazione l'utente potrà vivere nei panni di Robin, un ragazzo che, a causa delle forti pressioni di successo e per paura del giudizio degli altri, si chiude nella sua stanza, creandosi un mondo perfetto ma irreale.

Il secondo capitolo di questa tesi si concentra sullo stato dell'arte della VR e degli altri tipi di realtà esistenti.

Nel terzo capitolo verrà presentata l'applicazione LockedUp, mentre nel quarto verranno descritte tutte le teconolgie hardware e software utilizzate per questo progetto.

Nel quinto capitolo verrà spiegato in modo dettagliato con quali strumenti e come profilare un'app in Unity e come trovare gli eventuali problemi possono rallentare le prestazioni di un'applicazione; tutte le tecniche di ottimizzazione che possono essere utilizzate verranno descritte nel capitolo 6. I capitoli 7 e 8 condividono la stessa struttura e fanno riferimento rispettivamente all'atto 1 e all'atto 2 di LockedUp. Dopo aver descritto cosa succede e aver mostrato delle immagini del gameplay, verranno mostrati i risultati della prima profilazione e delle profilazioni effettuate dopo aver applicato ogni singola tecnica di ottimizzazione precedentemente descritta.

Capitolo 2

Cos'è la VR

La realtà virtuale (o VR, Virtual Reality) è una tecnologia che crea un ambiente virtuale che si sostituisce a quello reale. Utilizzando dispositivi come occhiali o visori VR, gli utenti possono immergersi in questo ambiente e interagire con esso.

La realtà virtuale combina grafica 3D avanzata, audio surround, feedback tattile e tracciamento dei movimenti per creare un'esperienza coinvolgente che può simulare la presenza fisica in un ambiente virtuale. Gli utenti possono esplorare e interagire con gli oggetti virtuali, spostarsi all'interno dell'ambiente e partecipare a eventi simulati.

La realtà virtuale viene utilizzata in diversi settori, tra cui l'intrattenimento, i videogiochi, la formazione, la medicina, l'architettura e il design, consentendo agli utenti di vivere esperienze che altrimenti sarebbero difficili, pericolose o impossibili da realizzare nel mondo reale.

I principali visori utilizzati oggi sono prodotti da Meta (Oculus) (fig. 4), Sony (Playstation VR) (fig. 2), HTC (fig. 1) e Samsung.

Figura 1: Visore HTC VIVE PRO 2 .

Figura 2: PlayStation VR .

2.1 Differenze con AR e MR

Ciò che differenzia la VR dalla realtà aumentata (AR) e dalla realtà mista (MR) è la completa mancanza, nel caso della VR, di elementi del mondo reale.

2.1.1 La Realtà Aumentata, AR

La realtà aumentata, o AR, è una tecnologia che combina elementi virtuali con l'ambiente fisico circostante, creando un'esperienza in cui gli oggetti digitali interagiscono con il mondo reale. A differenza quindi della realtà virtuale, che crea un ambiente completamente simulato, la realtà aumentata sovrappone elementi virtuali al mondo reale.

La realtà aumentata può essere sperimentata attraverso dispositivi come smartphone, tablet, occhiali AR o visori indossabili. Utilizzando la fotocamera del dispositivo e la tecnologia di rilevamento dell'ambiente circostante, la realtà aumentata è in grado di riconoscere e comprendere quest'ultimo e sovrapporrgli elementi virtuali.

Gli elementi virtuali nella realtà aumentata possono includere oggetti 3D, informazioni testuali, video, suoni o altri elementi interattivi. Ad esempio, un'app di realtà aumentata potrebbe consentire di puntare il proprio smartphone verso un edificio e visualizzare informazioni aggiuntive su di esso, come il nome, la storia o le recensioni.

2.1.2 La Realtà Mista, MR

La realtà mista, o MR, è una tecnologia che combina elementi della realtà virtuale (RV) e della realtà aumentata (AR) per creare un'esperienza in cui oggetti virtuali e reali coesistono e interagiscono tra loro.

La realtà mista consente agli utenti di vedere e interagire con oggetti virtuali all'interno del mondo reale in modo realistico e naturale. Questo viene reso possibile attraverso l'uso di dispositivi come visori o occhiali che integrano la realtà fisica con elementi virtuali in modo fluido.

A differenza della realtà aumentata, in cui gli oggetti virtuali vengono sovrapposti al mondo reale senza una reale interazione, la realtà mista offre una vera e propria fusione tra elementi virtuali e reali.

Ad esempio, utilizzando un dispositivo di realtà mista, si potrebbe vedere e interagire con un ologramma virtuale di un oggetto tridimensionale che si trova sul tavolo. Si può spostare l'oggetto virtuale, cambiarne la dimensione, ruotarlo e persino interagire fisicamente con esso.

Capitolo 3

LockedUp

LockedUp è un progetto ideato nel 2020 da un gruppo di studenti di Ingegneria del Cinema in collaborazione con il Politecnico di Torino e il National Film Board Of Canada.

LockedUp è un'esperienza di 20-25 minuti attraverso cui l'utente potrà immedesimarsi nella vita di Robin, un adolescente che, a causa della troppa pressione dell'avere successo nella vita e della paura del giudizio degli altri, si chiude nella sua stanza e nel suo proprio mondo.

La storia si divide in due atti e si svolge in due luoghi creati in CG: la classe, dove si svolge il primo atto, e la stanza di Robin, dove si svolge il secondo atto. I due atti verranno descritti successivamente in questo elaborato, nei capitolo (7.1 e 8.1).

Figura 3: Locandina di LockedUp .

Capitolo 4

Tecnologie Utilizzate

4.1 Hardware

4.1.1 Oculus Quest 2

L'applicazione è stata sviluppata per il visore Oculus Quest 2 di Meta.

Figura 4: Oculus Quest 2 .

Specifiche

Qui di seguito sono elencate le specifiche del visore Oculus Quest 2. [1]

- **Display**: L'Oculus Quest 2 ha un display LCD con una risoluzione di 1832 x 1920 pixel per occhio, offrendo una risoluzione totale di 3664 x 1920 pixel.

- **Refresh Rate**: Il refresh rate del display può essere impostato su 60 Hz o 90 Hz.
- **Processore**: Utilizza il chipset Qualcomm Snapdragon XR2, che offre prestazioni elevate e una migliore grafica rispetto al suo predecessore, l'Oculus Quest.
- **Memoria**: È disponibile in due varianti di memoria interna, 64 GB e 256 GB.
- **Audio**: L'audio integrato fornisce un'esperienza audio 3D immersiva.
- **Controller**: L'Oculus Quest 2 è dotato di due controller Oculus Touch che offrono tracciamento preciso dei movimenti delle mani e un'esperienza di gioco intuitiva.
- **Connessione**: Può essere utilizzato in modalità wireless senza bisogno di un PC collegato, ma può anche essere collegato a un PC tramite un cavo USB per sfruttare le funzionalità del PC VR.

Requisiti di prestazioni minime per un'app Oculus

Qui di seguito sono elencati i requisiti minimi per un'applicazione sviluppata per Oculus [2]; Il lavoro presentato in questa tesi si è contenctrato sull'ottimizzazione dell'applicazione "LockedUp" affinchè rispettasse tali requisiti senza compromettere gli effetti visivi.

In primo luogo, è importante garantire un numero adeguato di frame al secondo (FPS). Per un'applicazione non interattiva, si può mirare a 60 FPS, mentre per un'applicazione interattiva, come nel caso di "LockedUp", è necessario avere almeno 72 FPS come target.

Inoltre, un altro aspetto cruciale da considerare è il controllo delle chiamate alla GPU per il rendering degli oggetti e delle componenti grafiche. Queste chiamate, chiamate "draw call", devono essere mantenute sotto controllo poiché ogni GPU ha un limite gestibile di draw call. Superare tale limite potrebbe sovraccaricare la GPU e causare una perdita di FPS. Nel caso specifico di Oculus Quest 2, Meta suddivide le applicazioni in tre categorie:

- **Busy Simulation**: Per busy simulation si intende applicazioni con molti simulazioni, tanti personaggi o tanti NPC. Sono applicazioni di questo tipo i multiplayer FPS (First Person Shooter, sparatutto). Le applicazioni di questo tipo dovrebbero avere tra le 80 e le 200 draw calls.

- **Medium Simulation**: La maggior parte delle applicazioni ricade in questa categoria. Sono applicazioni non troppo popolate, come degli FPS single player o social games senza troppi utenti. Per queste applicazioni, Meta consiglia di rimanere tra le 200 e le 300 draw calls.
- **Light Simulation**: Applicazioni non troppo complesse dal punto di vista grafico, come per esempio applicazioni che utilizzano shader base con poche varianti. Fanno parte di questa categorie giochi come le escape rooms. Per questa categoria Meta consiglia di rimanere tra le 400 e le 600 draw calls.

LockedUp è classificabile tra le app Medium Simulation, quindi sarà importante non superare le 300 draw calls.

Platform	Draw Calls	Description
Quest 1	50-150	Busy Simulation
Quest 1	150-250	Medium Simulation
Quest 1	200-400	Light Simulation
Quest 2	80-200	Busy Simulation
Quest 2	200-300	Medium Simulation
Quest 2	400-600	Light Simulation

Figura 5: Tabella riassuntiva delle Draw Calls per ogni dispositivo e tipo di applicazione .

Altro parametro importante sono i triangoli da renderizzare: Meta, per l'Oculus Quest 2, consiglia di non superare i 750 mila - 1 milione di triangoli.

4.2 Software

4.2.1 Unity 3D

Unity 3D è un motore grafico sviluppato da Unity Tecnhologies per lo sviluppo di videogiochi e applicazioni interattive. Unity è disponibile sia per MacOS che Windows; la versione base, gratuita, è sufficiente per sviluppare progetti di qualità. Unity ha una serie di tool che facilitano l'importazione di modelli 3D da software come Blender, Cinema4D e Maya. Unity supporta le librerie Direct3D, OpenGL e OpenGL ES, permettendo di sviluppare app per Mac, Microsoft, Xbox, PlayStation, Linux, Web, Wii e vari tipi di visori, tra cui Oculus Quest 2. Dopo aver abbandonato il linguaggio UnityScript nel 2017, il linguaggio di scripting più usato in Unity è C#. Unity si appoggia per lo scripting a Visual Studio, una IDE di Microsoft che supporta IntelliSense, componente per l'autocompletamento e i suggerimenti. Unity si appoggia anche, su MacOS, a Xcode, la IDE di Apple.

4.2.2 XR Interaction Toolkit

XR Interaction Toolkit [3] è un sistema di interazione ad alto livello per creare app in VR e AR. Questo sistema è costituito da un Interaction Manager, da un set di Interactor e Interactable per gestire l'interazione e da un sistema di movimento, il Locomotion System. XR Interaciton Toolkit ha anche componenti che permettono di:

- Gestire vari controller XR (Oculus, OpenXR, Windows Mixed Reality)
- Gestire le azioni di hover, select e grab
- Restituire un feedback aptico attraverso i controller
- Interagire con XR Origin, una videocamera VR per la gestione di esperienze di questo tipo

Capitolo 5

Profiling di un'app Unity

5.1 Tools

5.1.1 Unity Profiler

Il profiler [4] è uno strumento di Unity fondamentale per ottenere informazioni sulle prestazioni delle proprie applicazioni e identificare eventuali colli di bottiglia che possono rallentare l'applicazione (figura 6). Utilizzando il profiler, è possibile ottenere informazioni sul tempo di esecuzione di ogni frame sia per la CPU che per la GPU, sull'utilizzo della memoria, sulle chiamate di disegno (draw call) e su altri parametri utili. Tutte queste informazioni vengono visualizzate attraverso grafici [5].

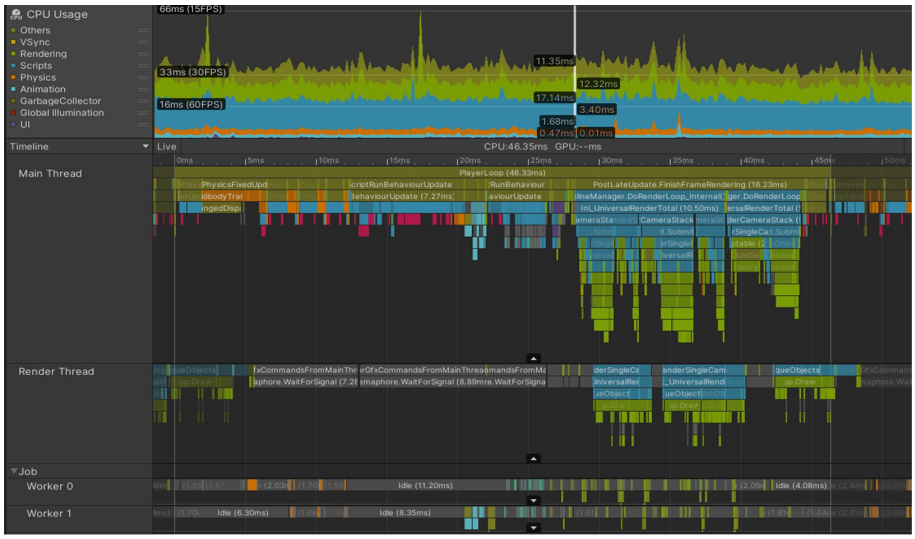


Figura 6: Unity Profiler .

E' possibile analizzare le informazioni raccolte attraverso due viste, la vista gerarchica (fig. 7) e la timeline (fig. 8); entrambe danno informazioni sul tempo di esecuzione di ogni processo e dei suoi sotto-processi. Il profiler si può utilizzare direttamente nel play mode di Unity, che è la soluzione più veloce ma sconsigliata se l'applicazione è destinata ad un dispositivo che non è il PC su cui si sta sviluppando l'app. Quando si utilizza il profiler in questa modalità, le informazioni raccolte mostrano le prestazioni dell'app sull'ambiente del PC. Per ottenere informazioni sulle prestazioni dell'applicazione sul dispositivo di destinazione, è necessario connettere il dispositivo al PC e eseguire una build speciale dell'applicazione per il dispositivo stesso. Successivamente, il profiler può essere collegato al dispositivo per raccogliere e analizzare le informazioni sulle prestazioni specifiche del dispositivo. Per accedere al profiler dobbiamo accedere al pannello **Window – Analysis – Profiler**.

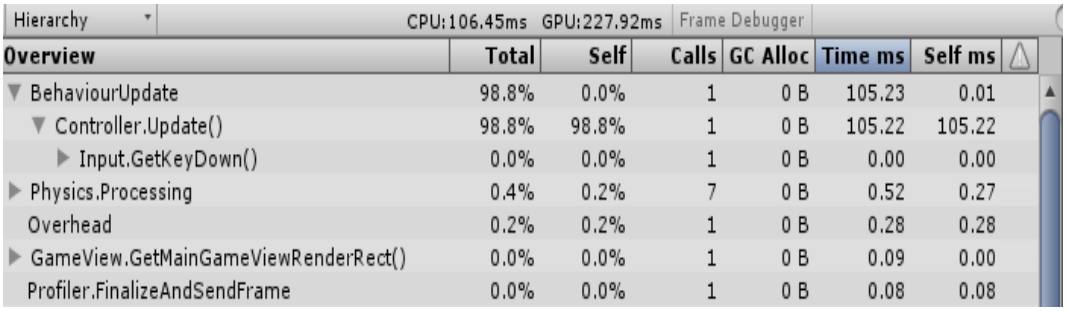


Figura 7: Vista gerarchica del Profiler .

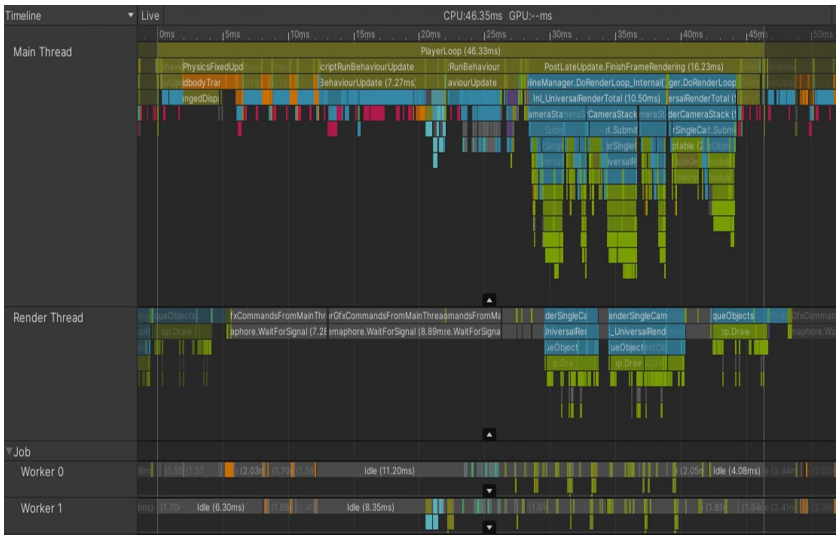


Figura 8: Timeline del Profiler .

5.1.2 Frame Debugger

Il frame debugger [6] è un tool che permette di fermare l'applicazione su un particolare frame e analizzarlo per ottenere la lista degli eventi di render che costituiscono il frame stesso (fig. 9). Dopo aver ottenuto la lista degli eventi, è possibile analizzare i passi di render uno ad uno e vedere in che stato si trovava il frame in quel momento per conoscere le informazioni riguardo al comportamento dei materiali, delle luci e degli oggetti in ciascun frame. [5].

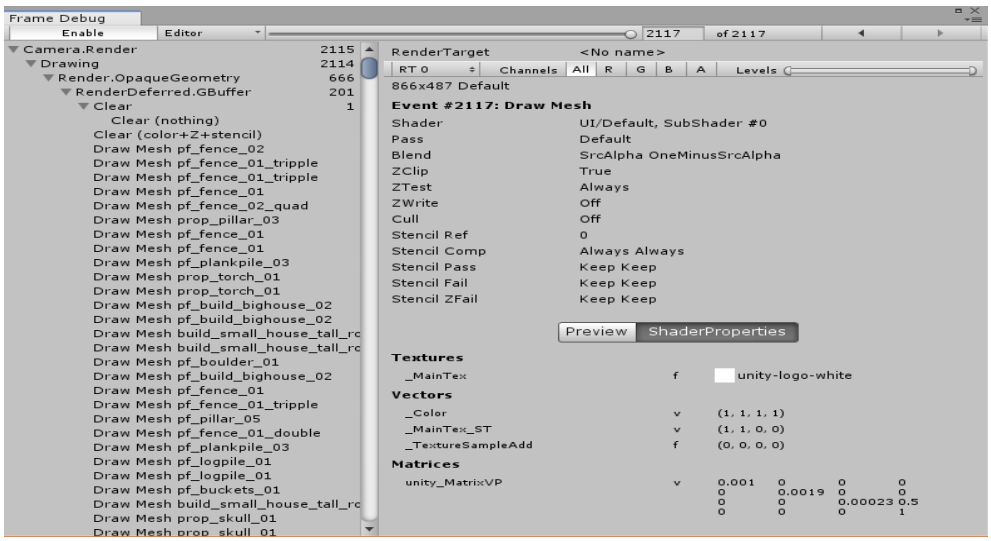


Figura 9: Schermata del Frame Debugger .

5.1.3 Render Doc

Render Doc è un frame debugger gratuito e open-source [7] che può essere usato per analizzare i singoli frame e offre funzionalità aggiuntive rispetto al frame debugger di Unity. Una delle caratteristiche distintive di RenderDoc è la possibilità di effettuare il profiling della GPU. Render Doc può analizzare frame di ogni applicazione basata su Vulkan, OpenGL, D3D11, OpenGL ES e D3D12. Una volta acquisito un frame, è possibile ottenere informazioni sui parametri cruciali per determinare se la GPU è il collo di bottiglia, come gli stadi della pipeline, le draw call, le texture map e il numero di chiamate per ciascun singolo shader. Grazie a queste informazioni, possiamo acquisire una comprensione precisa delle prestazioni della nostra GPU, dell'ordine di rendering e, soprattutto, del tempo impiegato da ogni oggetto per essere renderizzato e del numero di draw call coinvolte. Ciò ci consente di individuare situazioni in cui uno shader potrebbe richiedere troppi passaggi di rendering, sovraccaricando la GPU, o di identificare un'eccessiva quantità di draw

call per oggetti con lo stesso materiale. In tal caso, si possono adottare specifiche tecniche di ottimizzazione, che saranno descritte nel capitolo successivo6.

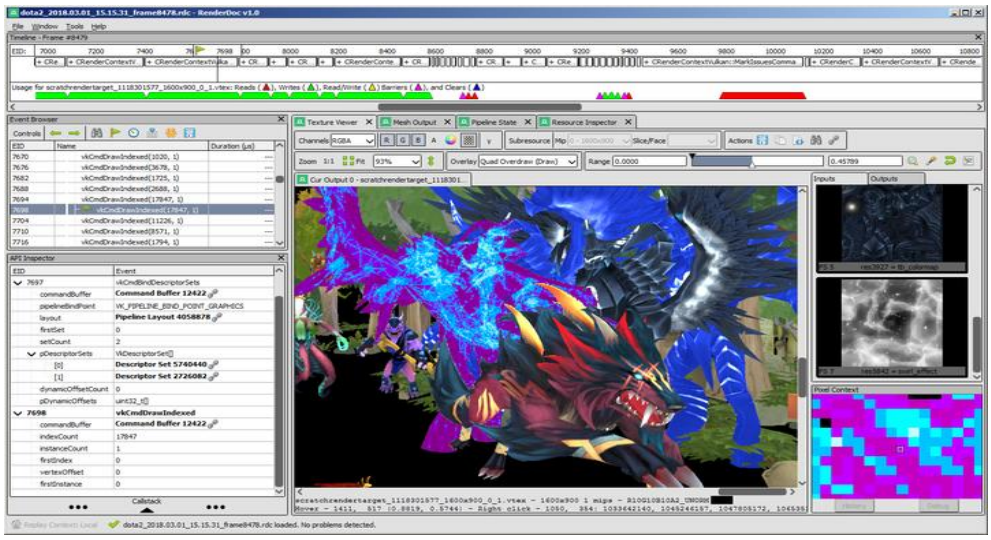


Figura 10: Schermata di RenderDoc .

5.1.4 OVR Metrics Tool

OVR Metrics [8] è un tool particolarmente comodo per profilare un'app in VR mentre la si sta testando su Oculus. Permette infatti di mostrare diversi dati utili, come il tasso di utilizzo di CPU e GPU e il frame rate in tempo reale, direttamente nell'Oculus mentre si sta testando l'app, in modo da non dover continuamente interrompere l'esperienza per controllare il profiler e avere un'idea immediata su come sta andando l'app (fig. 11).

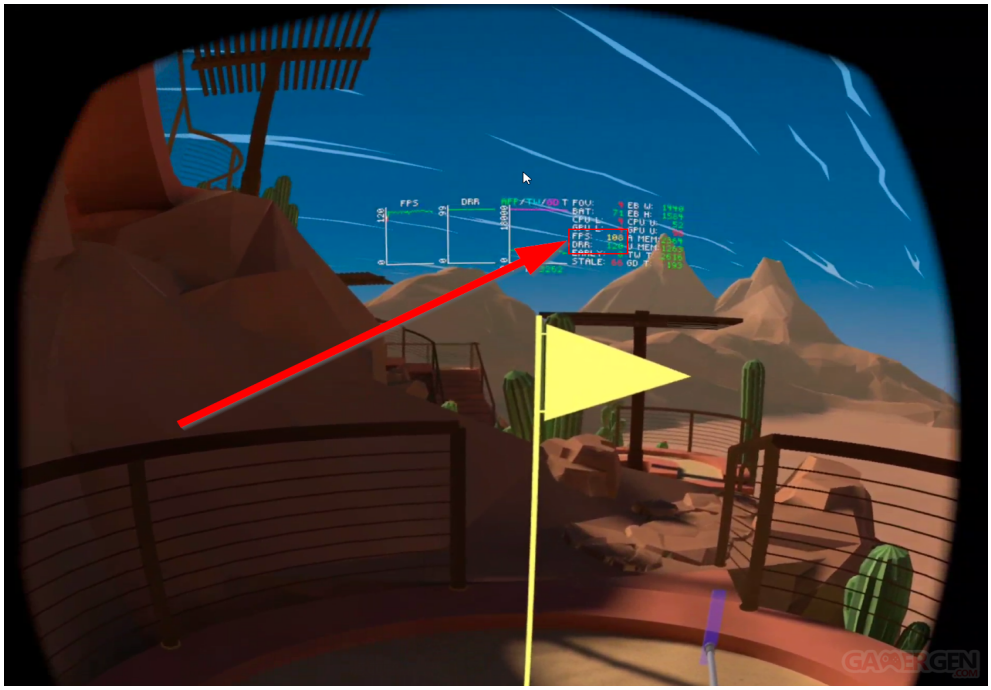


Figura 11: OVR Metrics Tool .

5.2 Come profilare un'applicazione

Delle buone prestazioni sono essenziali per creare esperienze immersive per gli utenti. In questo capitolo verrranno elencate le migliori tecniche per profilare un'applicazione e capire dove ci possono essere eventuali colli di bottiglia.

5.2.1 Tipi di profilazione

Ci sono due tipi principali di profiling: il sample-based profiling e l'instrumentation-based profiling. Una prima cosa da evidenziare è che ogni tecnica di profiling introduce un overhead tanto più grande tanto è maggiore il numero di informazioni che si vogliono raccogliere; i tempi di esecuzione analizzati col profiler potrebbero essere più alti di una decina di millisecondi di quelli "reali" a causa dell'overhead. Come si può verificare nella figura 47, relativa alla profilazione della prima scena di LockedUp, la stessa scena profilata con il Deep Profiling presenta il marker "ProfilingScope", che invece manca nell'analisi fatta con una profilazione più ad alto livello (figura 37).

Sample Based Profiling

Il sample-based profiling consiste nella raccolta di dati statistici sull'applicazione durante l'esecuzione. I profiler di questo tipo interrogano la call stack (una zona di memoria nella quale sono immagazzinate le informazioni sulle subroutine attive in un dato momento) ogni 'n' nanosecondi e usano queste informazioni per capire quando le funzioni sono state chiamate e per quanto tempo. Questo metodo è tanto più accurato quanto più si aumenta la frequenza di campionamento, ma rischia di introdurre un aumento dell' overhead [5].

Instrumentation-Based Profiling

Questo tipo di profiling consiste nell'aggiunta di profile markers nel codice. Questi marcatori raccolgono informazioni dettagliate sul tempo di eseuzione del codice in punti specifici del programma. Il profiler Unity descritto precedentemente è un profiler di questo tipo. I marker sono posizionati nella maggior parte delle API Unity per garantire un buon equilibrio tra raccolta delle informazioni e overhead [5]. Si possono anche inserire manualmente i marker in determinate parti di codice utilizzando l'API ProfilerMarker, come mostrato in figura 12 e 13.

```
using UnityEngine;
using Unity.Profiling;

public class ProfilerMarkerExample
{
    static readonly ProfilerMarker k_MyCodeMarker = new ProfilerMarker("My Code");

    void Update() {
        k_MyCodeMarker.Begin();
        Debug.Log("This code is being profiled");
        k_MyCodeMarker.End();
    }
}
```

Figura 12: Codice per inserire un profiler marker .

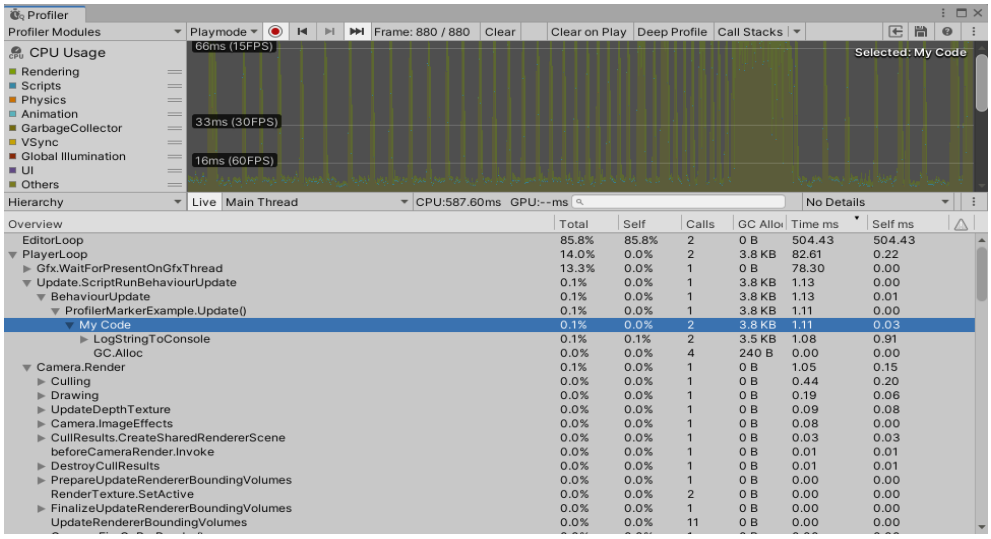


Figura 13: Il marker inserito nel codice tramite l'API viene mostrato nel profiler .

È importante notare che il profiler Unity offre anche una funzionalità chiamata Deep Profiling, che consente di ottenere un ulteriore livello di dettaglio nelle prestazioni del codice. Questo può essere utile per l'analisi approfondita di specifici aspetti delle prestazioni, ma può comportare un aumento dell'overhead.

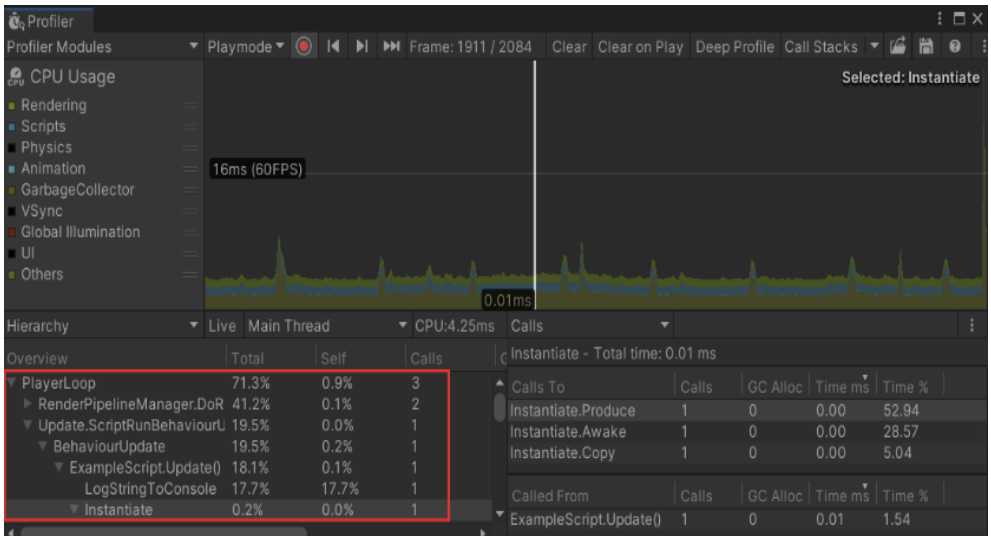


Figura 14: Esempio di marker inseriti nel codice .

5.2.2 Deep profiling

Unity di default effettua la profilazione solo sulle parti di codice che sono comprese tra due marker; questo include le chiamate alle funzioni native dell'engine, come i metodi Start e Update di MonoBehaviour. Gli altri campioni che possiamo visualizzare sono quelli relativi alle API di Unity che sono profilate. [9] Di norma vengono profilate tutte le API di Unity che possono introdurre overhead. Ad esempio, accedendo alla Camera tramite la API **Camera.main**, il profiler registrerà il campione **FindMainCamera**. Questo consente di identificare i punti in cui si sta facendo un uso intensivo di risorse o di API specifiche. Il deep profiling è una funzionalità avanzata che inserisce marcatori all'inizio e alla fine di ogni funzione nel codice dell'applicazione. Ciò consente di ottenere una maggiore dettagliatezza sulle prestazioni del codice, ma può comportare un aumento significativo dell'overhead e dell'utilizzo di memoria. Di conseguenza, Unity consiglia di utilizzare il deep profiling solo per applicazioni più piccole e meno complesse. Per usare il deep profiling nella build dell'applicazione, dobbiamo abilitarlo via **File – Build Settings – Deep Profiling Support**.

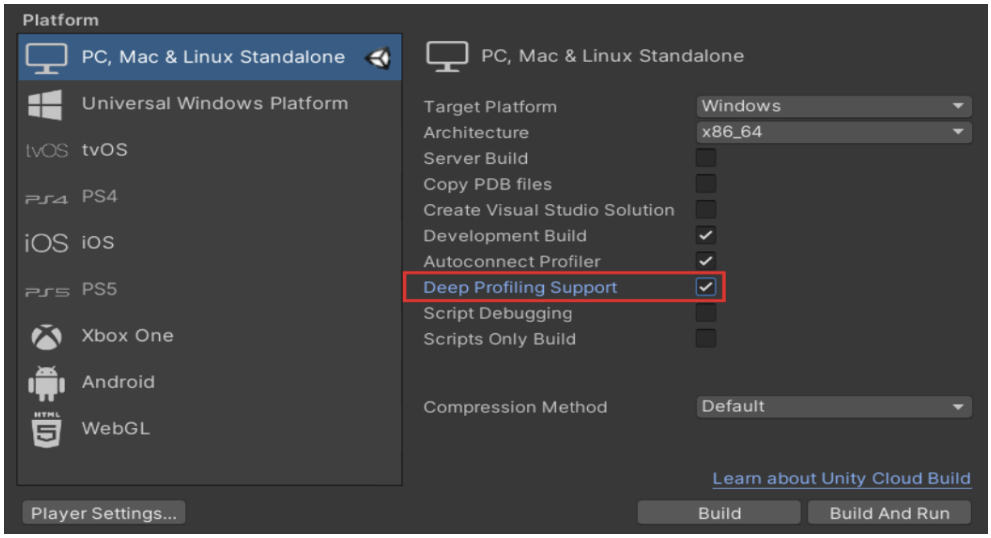


Figura 15: Esempio di marker inseriti nel codice .

5.2.3 Workflow

La prima cosa da fare è capire qual è il frame rate di riferimento: nel caso di un'applicazione per la realtà virtuale, gli fps minimi richiesti da meta sono 72. Ciò significa che ogni frame deve impiegare non più di 13 ms per essere renderizzato.

Il consiglio di Unity è quello di iniziare a fare il profiling già da subito, e di ripeterlo ogni volta che viene introdotta una grossa feature. Se si sta sviluppando

una applicazione per un altro dispositivo, è sempre necessario creare una **development build** collegando a essa il profiler. Per fare una prima profilazione ad alto livello si può partire con il deep profiling disattivo, e capire dove c'è un tempo di CPU o GPU troppo alto. Se le informazioni raccolte in questo modo non sono sufficienti per capire il problema, è necessario attivare il deep profiling, tenendo comunque a mente che questa modalità di profiling rallenta i tempi di esecuzione [5].

Il flowchart (fig. 16 illustrato qua sotto è un'ottima guida per capire come eseguire un corretto profiling.

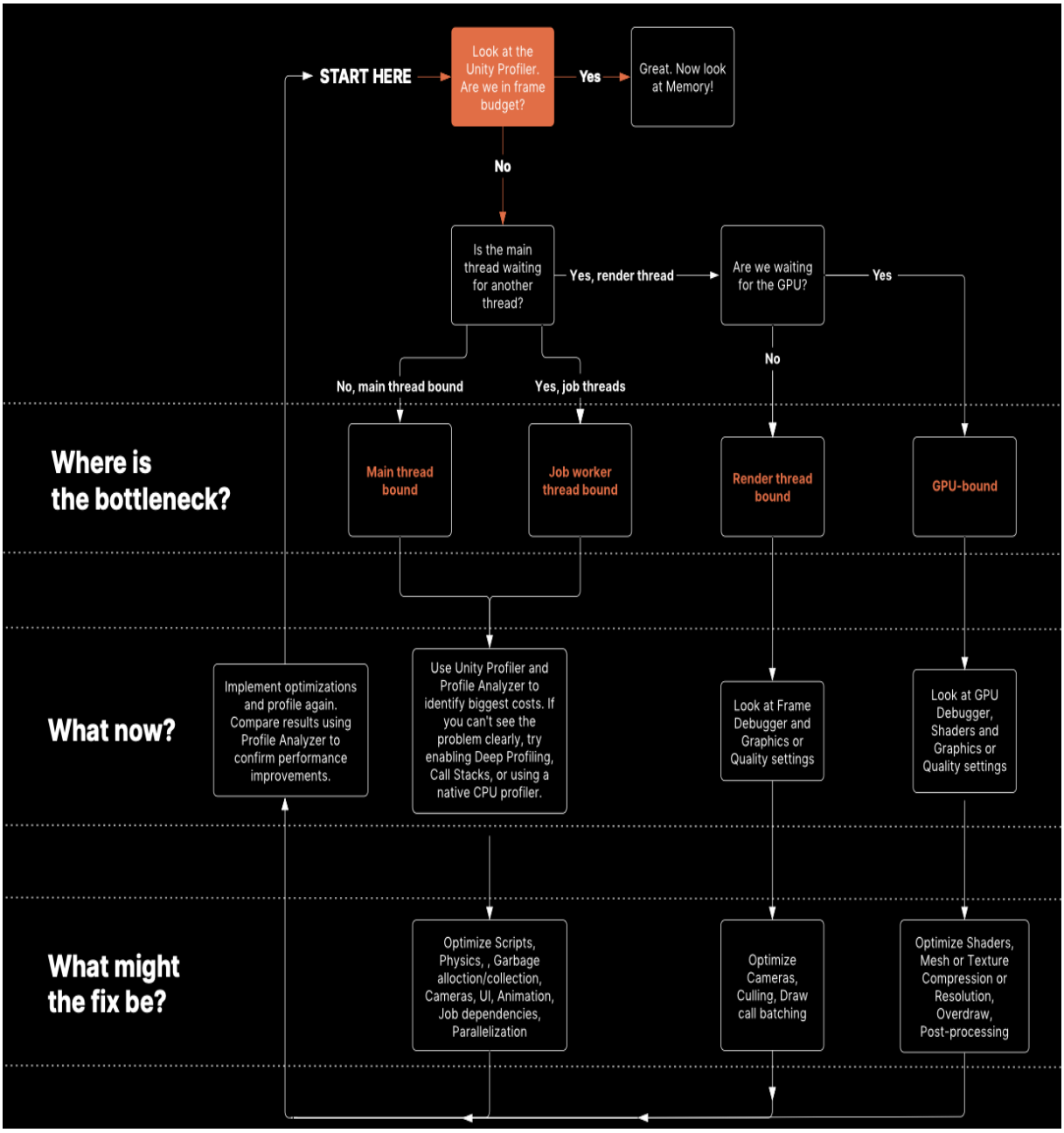


Figura 16: Flowchart per eseguire il profiling .

Da questo flowchart emerge come i colli di bottiglia possono trovarsi nella CPU (ovvero nel main thread, nel render thread o nei job worker thread) e nella GPU. Nel caso di collo di bottiglia nella CPU si parlerà di applicazione CPU-bound, nell'altro caso di applicazione GPU-bound.

5.2.4 CPU Bound

Se si riscontrano tempi di esecuzione elevati lato CPU, è importante individuare il collo di bottiglia specifico per identificare l'area del codice che sta causando il rallentamento.

È vero che le CPU moderne hanno più core che possono lavorare in modo indipendente e contemporaneo. Pertanto, è improbabile che l'intera CPU sia il collo di bottiglia. Spesso il problema si trova in uno specifico thread o in una determinata sezione del codice che richiede un carico di lavoro eccessivo.

In un'applicazione Unity, i tre thread principali di riferimento sono [5]:

- **Main Thread**: è dove viene svolta la parte logica dell'applicazione inclusa la fisica, le animazioni, la UI e il rendering. Per ottimizzare questo thread, spesso si deve guardare la fisica, l'update degli script della classe MonoBehaviour, garbage collection, culling, batching e le animazioni. Questi aspetti verranno trattati in modo più approfondito nel capitolo 6.

- **Render Thread**: Durante il rendering, il main thread esamina la scena, si occupa del culling e del batch delle draw call (tecniche di ottimizzazione approfondite nel prossimo capitolo) e determina quali oggetti devono essere renderizzati creando una lista di rendering. Questa lista viene passata al render thread, che comunica con le API grafiche che si interfacciano con la GPU.

- **Job Worker Threads**: Unity offre la possibilità di utilizzare i Job Worker Threads per distribuire il lavoro su thread separati e ridurre il carico sul Main Thread. I job worker permettono di usare le prestazioni perchè sfruttano tutti i core della CPU e dividono il lavoro in più compiti da svolgere in parallelo, anzichè svolgere tutto su un unico core, come mostrato in figura 17. [10] [11]

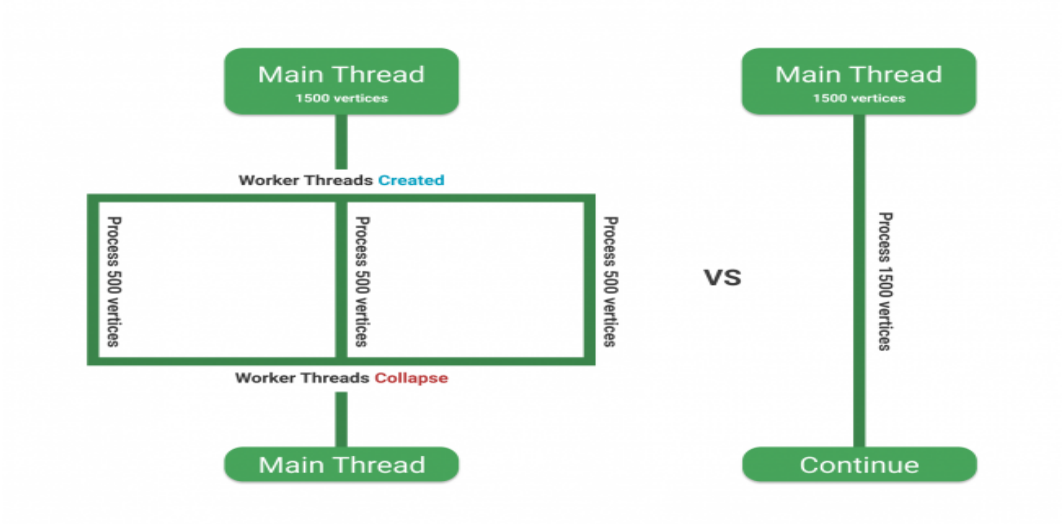


Figura 17: Funzionamento dei Job Worker Threads .

5.2.5 GPU Bound

Se il Main Thread spende troppo in marker come "Gfx.WaitForPresentOnGfxThread" e il render thread nel frattempo è bloccato su marker come "Gfx.PresentFrame" significa che l'applicazione è GPU-Bound [5].

In questo caso, i problemi comuni da risolvere sono [5]:

- Effetti di post processing particolarmente dispendiosi.
- Shader con troppi frammenti a causa della logica di batch utilizzata inefficiente.
- Un overdraw, ovvero troppi pixel disegnati sovrapposti nella coda del transparent render.
- Risoluzione dello schermo troppo alta.
- Nessun utilizzo di LOD (tecnica di ottimizzazione che verrà descritta nel prossimo capitolo).

Capitolo 6

Ottimizzazione di un'app Unity

In questo capitolo verranno elencate le possibili tecniche di ottimizzazione per un'applicazione Unity.

6.1 LOD

Il LOD (Level Of Detail) è una tecnica che consiste nell'ottimizzare una mesh in base alla distanza dalla camera. Di default infatti un oggetto ha la stessa mesh a prescindere dalla distanza dalla camera: quindi, un oggetto che ha un elevato numero di triangoli, verrà renderizzato allo stesso modo sia che l'utente lo veda davanti a se, sia che l'utente sia lontano da esso, vedendolo solo in lontananza. Ovviamente renderizzare gli oggetti di una scena in questo modo porta a un grande spreco di risorse, per cui vengono utilizzati diversi livelli di dettaglio. Per ogni oggetto vengono create diverse versioni della mesh originaria, con un numero sempre decrescente di triangoli: in base alla distanza tra l'oggetto e la camera, verrà visualizzata una versione diversa della mesh. Le mesh si possono ottimizzare utilizzando software di modellazione 3D come Blender. Due esempi di utilizzo di LOD si possono vedere nelle figure 18 [12] e 19 [13].

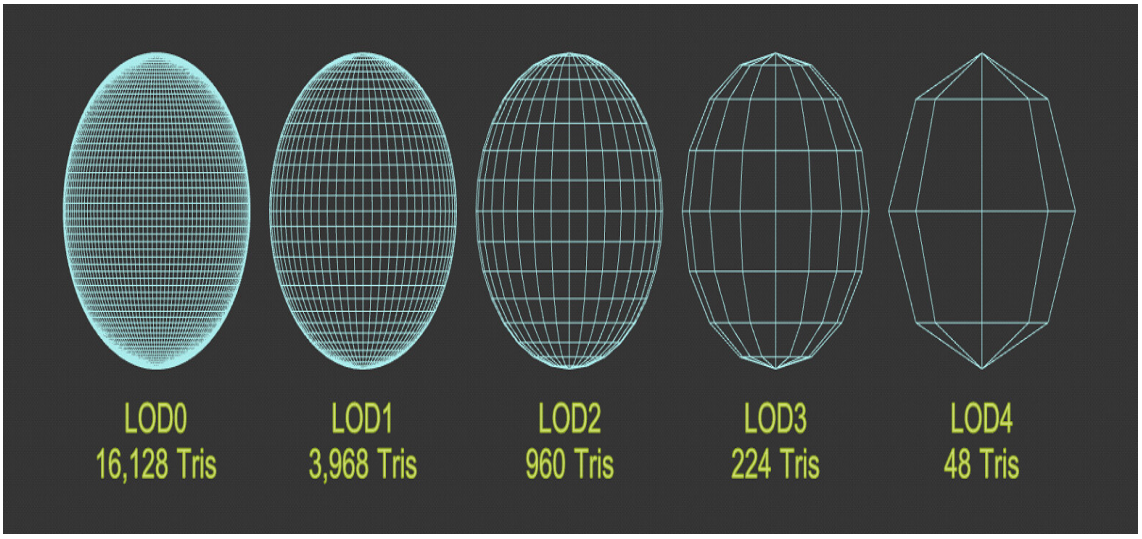


Figura 18: Esempio di LOD .

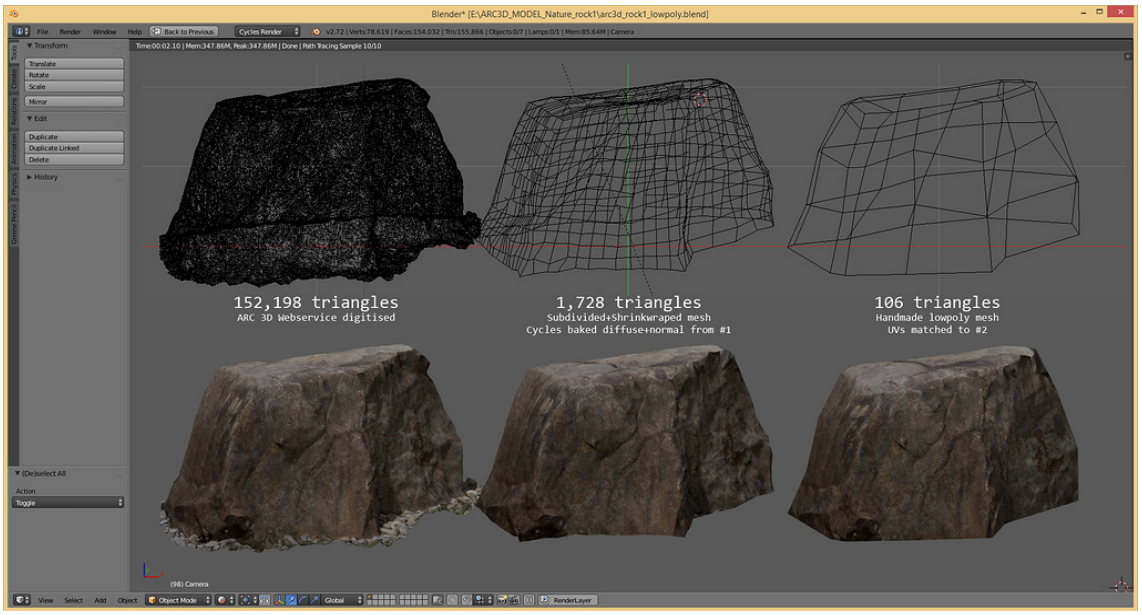


Figura 19: Esempio di LOD .

Per utilizzare i LOD su Unity [14] bisogna inserire nel GameObject che si vuole ottimizzare il componente "LOD Group". Questo componente controlla il comportamento dei vari livelli e permette di impostare le distanze alle quali avvengono le transizioni tra le mesh. Le transizioni sono molto importanti per evitare che l'utente veda chiaramente il passaggio da una mesh all'altra (questa

situazione è chiamata popping) [15]. Per rendere le transizioni più morbide, Unity permette di utilizzare il cross-fade: nello stesso istante vengono renderizzati due livelli, assegnando un peso da 1 a 0 al primo livello e da 0 a 1 al secondo livello (fig. 20)

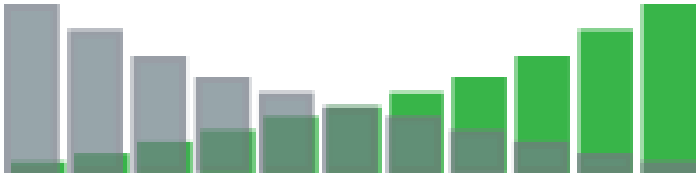


Figura 20: Cross-fade tra due livelli .

Un altro tipo di transizione offerto da Unity è il modello Speed Tree. Speed Tree è un software di modellazione creato da Interactive Data Visualization usato per generare modelli di alberi e fogliame utilizzato molto nelle animazioni e nei videogiochi [16] [17] [18].

Le geometrie di questo tipo memorizzano la posizione di ogni vertice nel LOD successivo; quindi per ogni vertice sono disponibili le informazioni per effettuare un'interpolazione tra la sua posizione precedente e la sua posizione successiva. Unity renderizza solo il LOD utilizzato in un dato momento, utilizzando un valore tra 0 e 1 per indicare al vertice se e quanto muoversi verso la posizione successiva.

In generale, la finestra del LOD Group Component su Unity si presenta così (fig. 21).

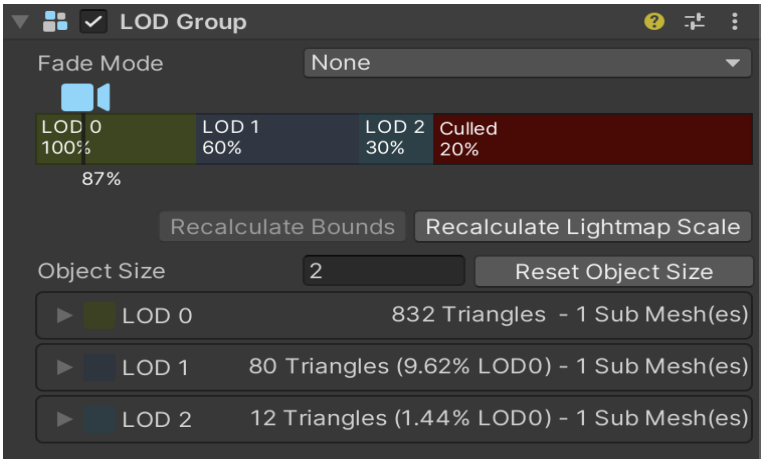


Figura 21: LOD Group Component .

Tramite il menù a tendina "Fade Mode" è possibile scegliere se utilizzare una transizione cross-fade, una speed tree o nessuna. Trascinando le etichette dei vari LOD è possibile modificare la distanza alla quale avviene la transizione tra un LOD e l'altro.

6.2 Draw Call Batching

Il batching è una tecnica utilizzata per ridurre il numero di draw call, parametro fondamentale per le prestazioni di un'applicazione, come già visto nel paragrafo 4.1.1.

Le draw call che contengono pochi vertici sono renderizzate velocemente dalla GPU, ma spesso costituiscono un problema perchè impediscono alla CPU di utilizzare la GPU a pieno, creando dei tempi d'attesa.

Unity quindi permette di utilizzare il batching, ovvero raggruppare le draw call degli oggetti che condividono lo stesso materiali, abbassando i tempi di processamento della CPU. [19] [20]

Utilizzando il batching è importante non unire le draw call di oggetti distanti tra di loro, per non rischiare di rendere inutili i benefici dell'occlusion culling, tecnica che verrà trattata più avanti in questo capitolo (paragrafo 6.5).

Unity offre due tipi di batching: batching statico e batching dinamico.

6.2.1 Static Batching

Il batching statico combina le mesh degli oggetti statici per ridurre le draw calls. Quello che fa Unity è partire da diverse mesh con lo stesso materiale e creare un'unica grande mesh; in questo modo verranno utilizzate meno draw calls ma ci sarà un maggior impiego della memoria. Infatti, se abbiamo 10 mesh che condividono lo stesso materiale, Unity dovrà creare 10 copie di quelle mesh per poi creare una mesh unica. Un batch di Unity può contenere al massimo 64.000 vertici; se ci sono più vertici, verrà creato un altro batch. [21]

Per attivare lo static batching è necessario selezionare gli oggetti che vogliamo includere nel batch e attivare la spunta "Batching Static" dall'Inspector.

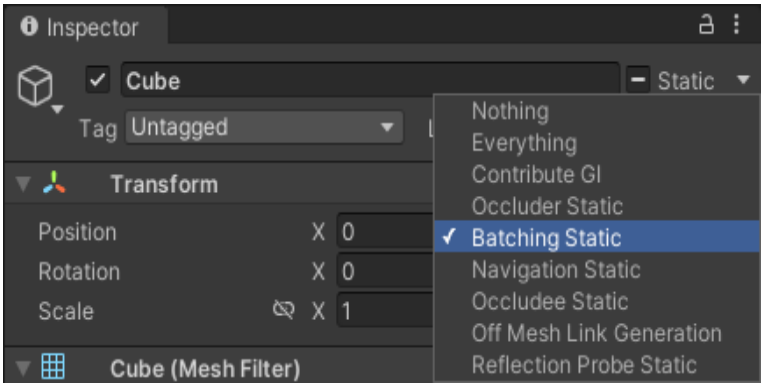


Figura 22: Attivazione Static Batching .

6.2.2 Dynamic Batching

Il Dynamic Batching è utilizzato per tutti quei GameObject che si muovono nella scena. Il dynamic batching è una forma di batching in cui gli oggetti vengono raggruppati dinamicamente prima del rendering. Invece di inviare una draw call separata per ogni oggetto, gli oggetti con caratteristiche simili o identiche vengono elaborati dalla CPU per creare un unico batch, che poi viene renderizzato in una singola draw call. Il batching dinamico trasforma tutti i vertici della mesh a cui è applicato nelle coordinate del mondo utlizzando la CPU anzichè la GPU; per questo il dynamic bathcing è una ottimizzazione se il lavoro necessario per trasformare i vertici(cioè posizionarli e orientali correttamente nello spazio 3D) è minore del costo della draw call.

Il dynamic batching non può essere applicato a mesh con più di 225 vertici e con più di 900 attributi per i vertici. Come per lo static batching, il dynamic batching può essere attivato dall'inspector e funziona solo con mesh che condividono lo stesso materiale. [**?**]

6.3 Combine Meshes

Combine Meshes è un metodo utile per prendere diverse mesh che condividono lo stesso materiale e la stessa texture e creare una sola mesh. Il combine meshes gode di una maggiore flessibilità rispetto al batching statico.Il secondo,infatti,è una tecnica di ottimizzazione automatica di Unity, mentre il combine meshes permette di scegliere quali mesh combinare secondo specifici criteri Ovviamnte va trovato un trade-off tra creare una mesh unica per vari oggetti distanti e utilizzare il culling. [22]

6.4 GPU Instancing

GPU Instancing è una tecnica di ottimizzazione per le draw call che renderizza più copie di una mesh con lo stesso materiale in una sola draw call. Questa tecnica è utile per gli oggetti che compaiono diverse volte in una scena, come alberi o cespugli. Questa tecnica funziona solo con oggetti che hanno la stessa mesh e lo stesso materiale. [23] Per utilizzarla:

- Lo shader del materiale deve supportare il GPU Instancing.
- La mesh deve essere di tipo Mesh Renderer; in questo caso Unity aggiunge la mesh a una lista e poi verifica se può effettuare l'instancing. Il GPU instancing non funziona sulle Skinned Mesh Renderer, usate spesso per le persone (e utilizzate per tutti gli studenti nella prima scena di LockedUp).

Per attivare il GPU Instancing è necessario selezionare un materiale, andare nell'inspector e attivare la spunta relativa a questa opzione.

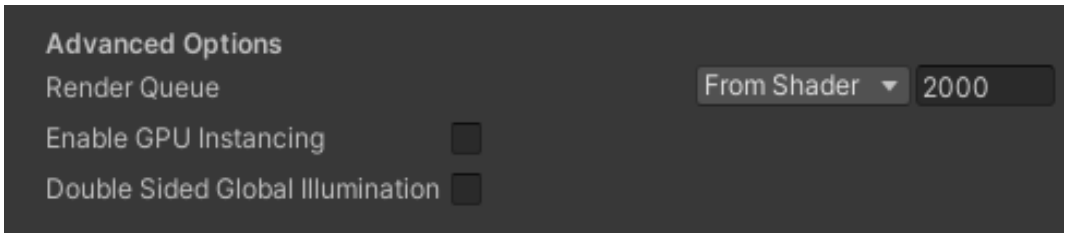


Figura 23: Attivazione GPU Instancing .

Se un oggetto è segnato come "statico" Unity proverà a fare il batch: se questo ha successo, verrà ignorato il GPU Instancing. Su quale sia il migliore dei due non c'è una risposta esatta, dipende dalla situazione in cui ci si trova, quindi conviene sempre testare entrambi i metodi e profilarli per vedere quale funziona meglio.

6.5 Culling

Il culling è una tecnica utilizzata per renderizzare solo gli oggetti visibili dalla camera in un dato momento. Ogni frame la camera analizza la scena ed esclude gli oggetti che non dovranno essere disegnati. Ci sono due tipi di culling: l'occlusion culling e il frustum culling. Il frustum culling permette di non disegnare tutti gli oggetti che non sono nel cono visivo della camera, ma utilizzando solo questo tipo di culling vengono comunque disegnati tutti gli oggetti che sono nascosti da altri oggetti, e quindi non visibili: per questo viene utilizzato anche l'occlusion culling. La differenza tra i due tipi di culling è ben visibile nelle figure 26 e 27.

Per attivare il culling bisogna prima di tutto marcare ogni oggetto come occluder e/o occludee (un occluder potrà coprire un altro oggetto mentre on occludee potrà essere coperto da un altro oggetto) tramite l'inspector. [24]

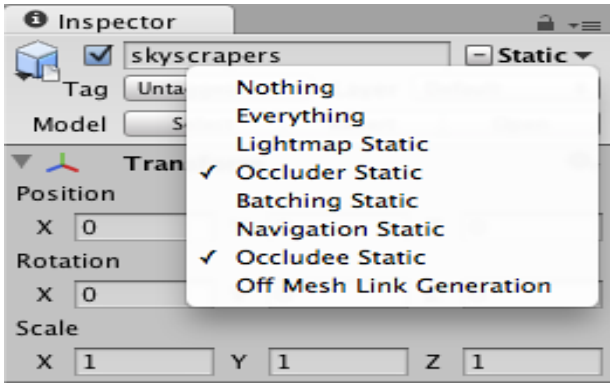


Figura 24: Attivazione Culling .

A quel punto, nella finestra di culling, si potrà effettuare il bake dei dati.

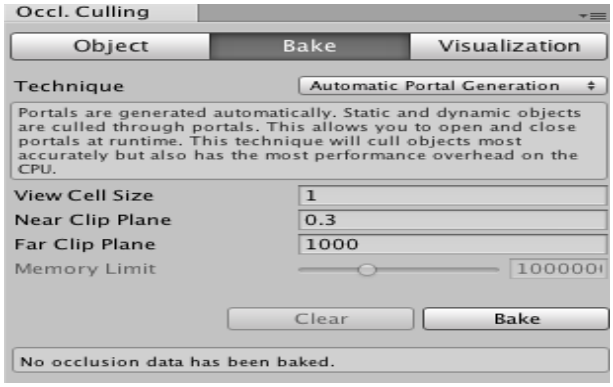


Figura 25: Baking delle mappe di occlusione .

Il baking consiste nel generare dei dati relativi alle occlusioni nell'editor e poi utilizzarli nell'app a runtime. Quando viene effettuato il baking, Unity divide la scena in celle e genera dati relativi alla geometria contenuta nelle celle e la visibilità tra celle adiacenti. Unity si occuperà poi di caricare questi dati a runtime e verificare in ogni istante quali oggetti sono visibili.

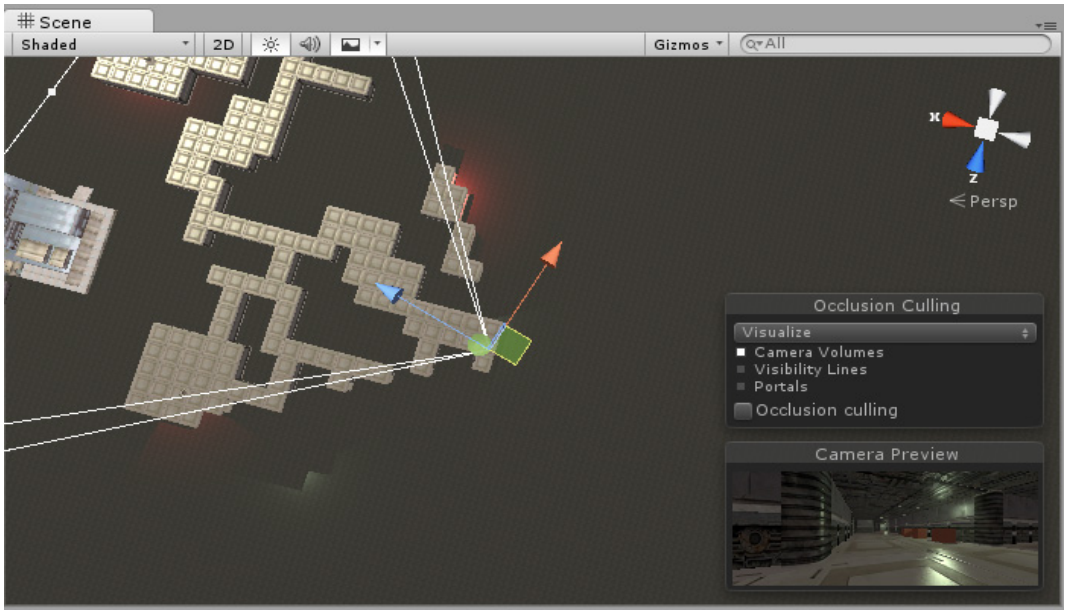


Figura 26: Frustum Culling .

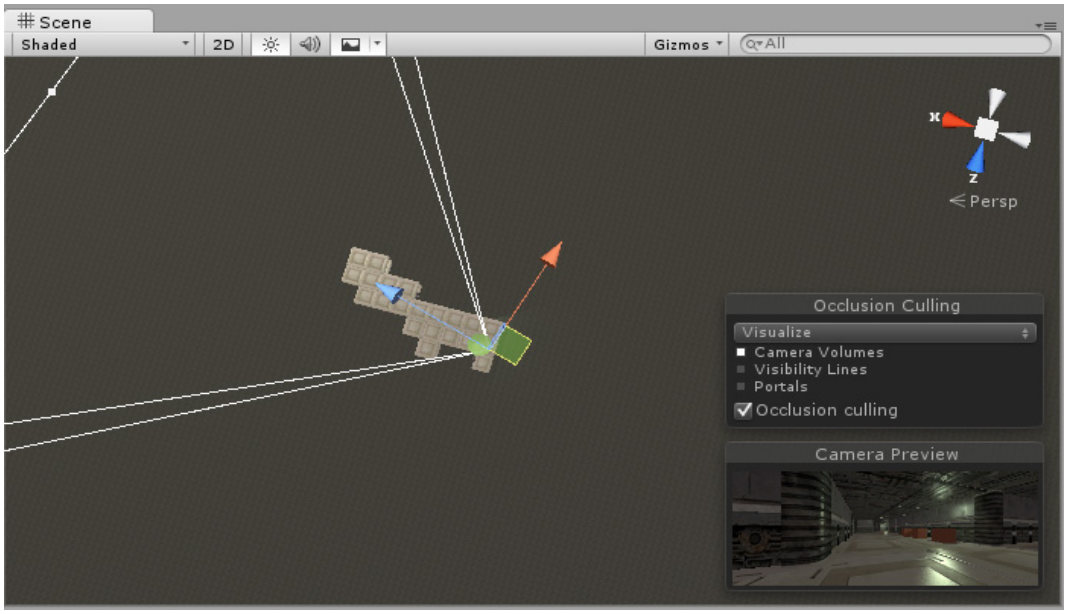


Figura 27: Occlusion Culling .

6.6 Baking delle Luci

L'illuminazione realtime può essere molto pesante per un PC, quindi per un'app destinata a Oculus è indispensabile ottimizzare anche questo aspetto. Il baking delle luci consiste nel pre-calcolo delle luci e delle ombre per una scena; a quel punto Unity sa dove illuminare e dove lasciare in ombra un oggetto dando l'impressione che ci sia una effettiva illuminazione. A runtime il renderer deve solo controllare tutte queste informazioni nella mappa generata (esattamente come succede per il culling). [25] Il rovescio della medaglia è che avremo bisogno di più spazio per archiviare tutte le lightmap relative a texture e oggetti, ma le prestazioni aumenteranno notevolmente.

Figura 28: Baking delle luci .

Unity permette anche di utilizzare un misto di luci real-time e luci baked.

Per utilizzare le luci baked, è necessario prima indicare a Unity che le luci in scena saranno di questo tipo, selezionando la luce e scegliendo questa modalità nell'inspector, e successivamente generare le lightmap nella sezione lighting. Per generare lightmap anche per le mesh nella scena, è necessario selezionare il GameObject, andare nell'inspector e sotto "Mesh Renderer" attivare la casella per la generazione di lightmap.

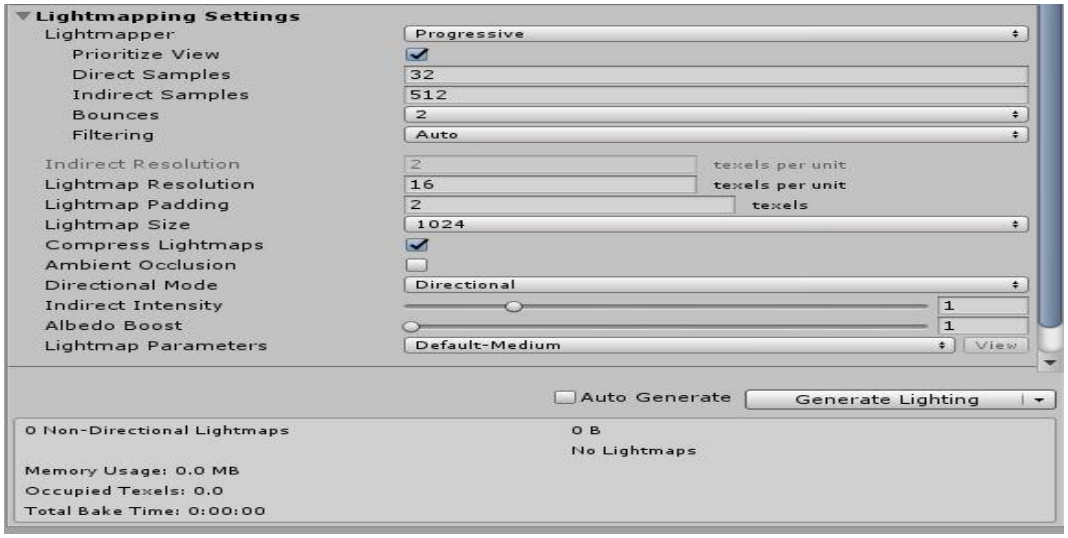


Figura 29: Generazione delle lightmap .

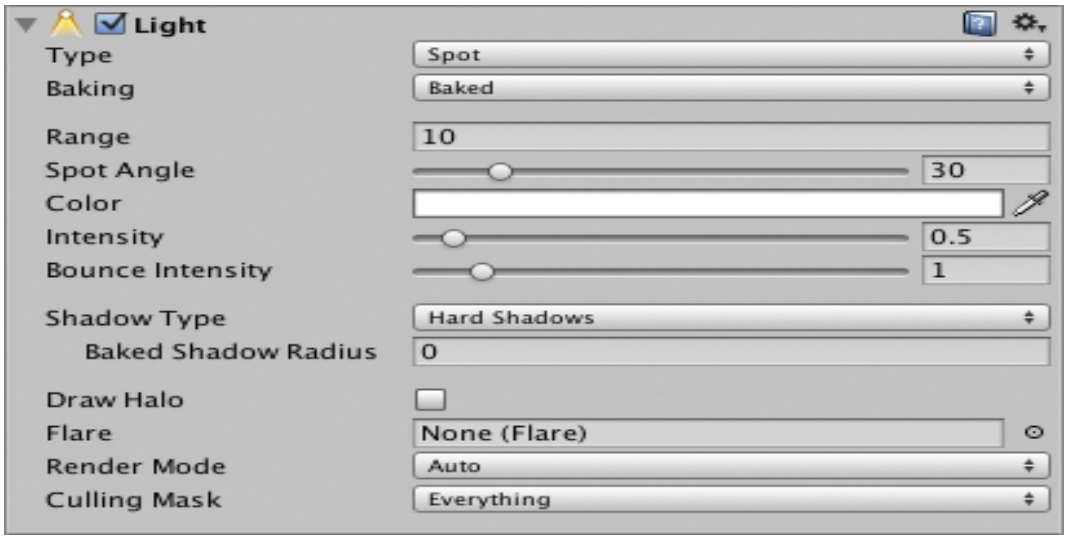


Figura 30: Light Inspector .

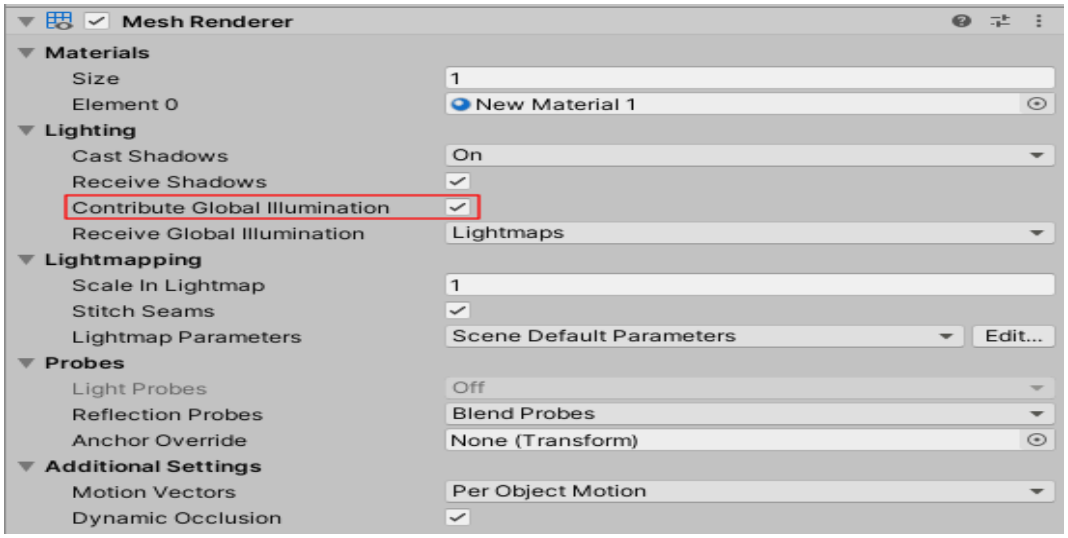


Figura 31: Attivare le lightmap sulle mesh .

6.7 Fisica

Un primo accorgimento da adottare per ottimizzare la parte fisica di un'app Unity è la sostituzione dei Mesh Collider. I mesh collider sono dei collider che si "adeguano" alla forma della mesh corrispondente, garantendo assoluta precisione nella gestione delle collisioni, ma sono anche dispendiosi, perciò è meglio sostituirli con dei collider più approssimati di forma cubica, sferica o cilindrica. L'engine fisico si aggiorna in determinati istanti temporali: si possono modificare andando in **Edit – Project Settings – Time**. Il campo "Fixed Timestep" definisce l'intervallo di tempo utilizzato dall'engine fisico per aggiornarsi. Questo valore va scelto estremamente bene: se un frame impiega troppo tempo per essere preparato, per esempio 40ms, e il timestep utilizzato è di 20 ms, avremo due simulazioni fisiche sullo stesso frame, il che rallenta ulteriormente l'elaborazione. Si può impostare un **Maximum Allowed Timestep**; se gli update fisici superano questo limite, unity inizia a ridurli. Per ridurre i problemi con la fisica si può:

- Ridurre la frequenza di simulazione. In particolare per i dispositivi di fascia bassa, è consigliabile portarla a poco più del frame rate (per esempio, a 0.035 secondi se vogliamo andare a 30 fps).
- Ridurre il Maximum Allowed Timestep per far si che ci siano meno update fisici in un frame

Un altro aspetto da controllare è il raycasting: troppi raggi infatti possono appesantire l'app. Se non possiamo ridurre il numero di raggi, possiamo quantomeno

usare RaycastCommand [26], una struttura che permette di eseguire un comando di raycast in modo asincrono affidandolo ai Job System, spostano il carico dal main thread ai sub-thread [27].

Quando si ha una collisione e viene chiamato un metodo come MonoBehaviour.OnCollisionEnter o OnCollisionStay, viene generata una callback e per ognuna di esse viene creato l'oggetto che ha generato la collisione che viene passato come argomento della callback, obbligando il garbage collector a rimuovere ogni oggetto. Abilitando nelle impostazioni di progetto l'opzione reuseCollisionCallbacks, per ogni callback viene creata solo uan istanza di collisione che viene poi riutilizzata. [28].

Capitolo 7

LockedUp, Atto 1

7.1 Introduzione

All'inizio del primo atto di LockedUp l'utente si ritrova in un ambiente nero e vede davanti a se il banco del protagonista di questa storia, Robin (fig. 32). Dopo essersi avvicinato al banco ed aver interagito con il walkman, intorno a se comparirà la classe di Robin, con i suoi compagni e la sua insegnante (fig. 33).

Figura 32: LockedUp, atto 1, banco di Robin .

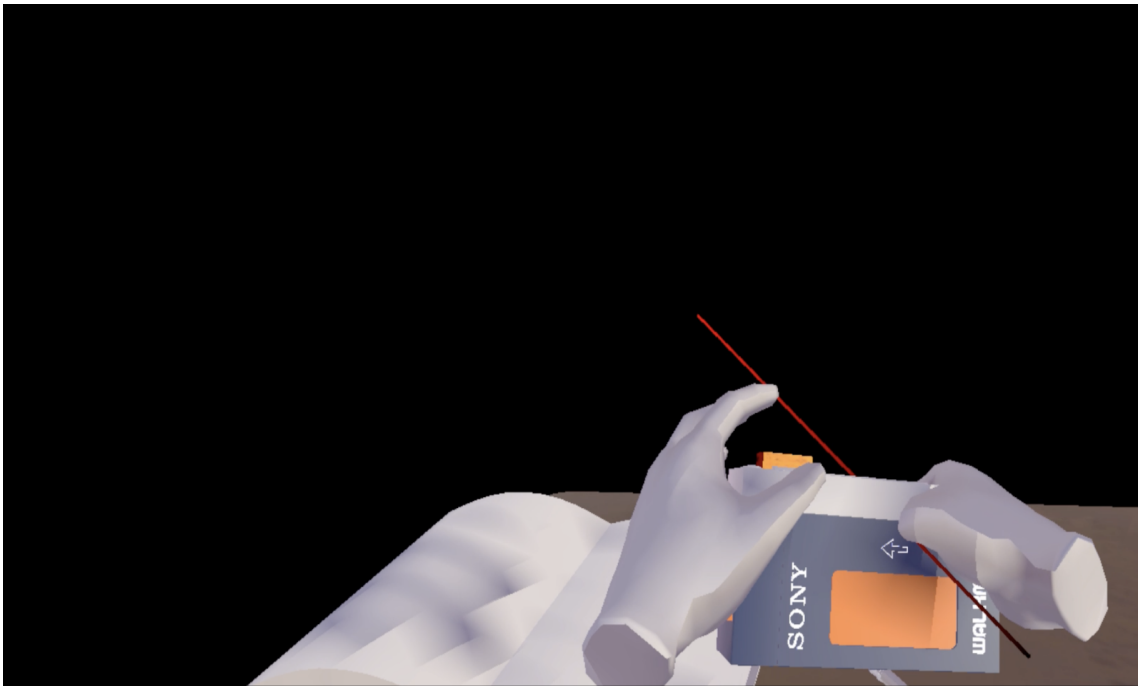


Figura 33: LockedUp, atto 1, interazione con il Walkman .

Una volta in classe (fig. 34), Robin verrà chiamato dalla sua insegnante a risolvere una equazione alla lavagna, ma, nel momento in cui l'utente prenderà in mano il gessetto e inizierà a scrivere, le equazioni inzieranno a cambiare velocemente (fig. 35).

Figura 34: LockedUp, atto 1, classe .

Figura 35: LockedUp, atto 1, interazione con la lavagna .

Dopo vari tentativi, l'insegnante, spazientita, rimprovera Robin, e i suoi compagni ridacchiano prendendolo in giro. Robin si sentirà quindi sopraffatto dal peso si questo fallimento e avrà un attacco di panico: l'ambiente inizierà a diventare via via nero e a tremare (fig. 36). L'effetto è dato da uno shader applicato all'ambiente.

Figura 36: LockedUp, atto 1, attacco di panioc .

Dato che i due ambienti sono abbastanza diversi tra di loro, qui di seguito verranno mostrate le profilazioni sia per la prima scena, in cui c'è solo il banco di Robin, sia per la seconda scena nella classe.

7.2 Profilazione

Per avere una visione ad alto livello del problema, senza avere problemi di overhead, è stata effettuata una prima profilazione alla prima scena senza Deep Profiling.

Di seguito verranno illustrati i risultati delle analisi sia della prima parte dell'atto 1, quando l'ambiente è nero e davanti all'utente c'è solo il banco di Robin con il walkman, sia della seconda parte, quando l'utente si trova dentro la classe con i compagni di Robin e l'insegnante.

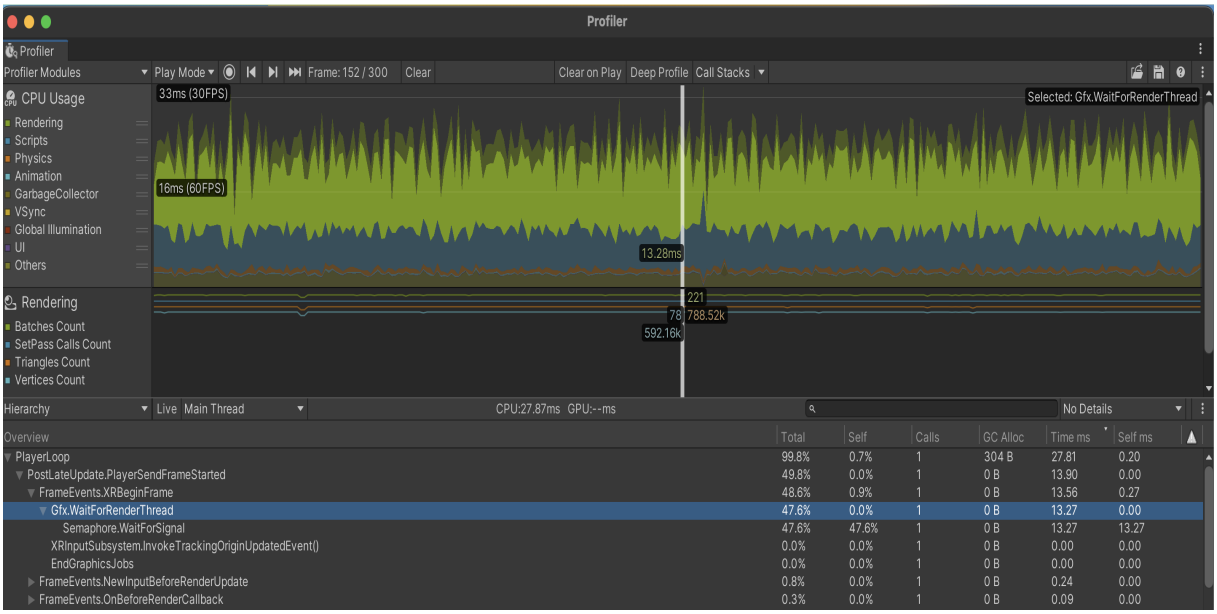


Figura 37: Profilazione generale della prima parte .

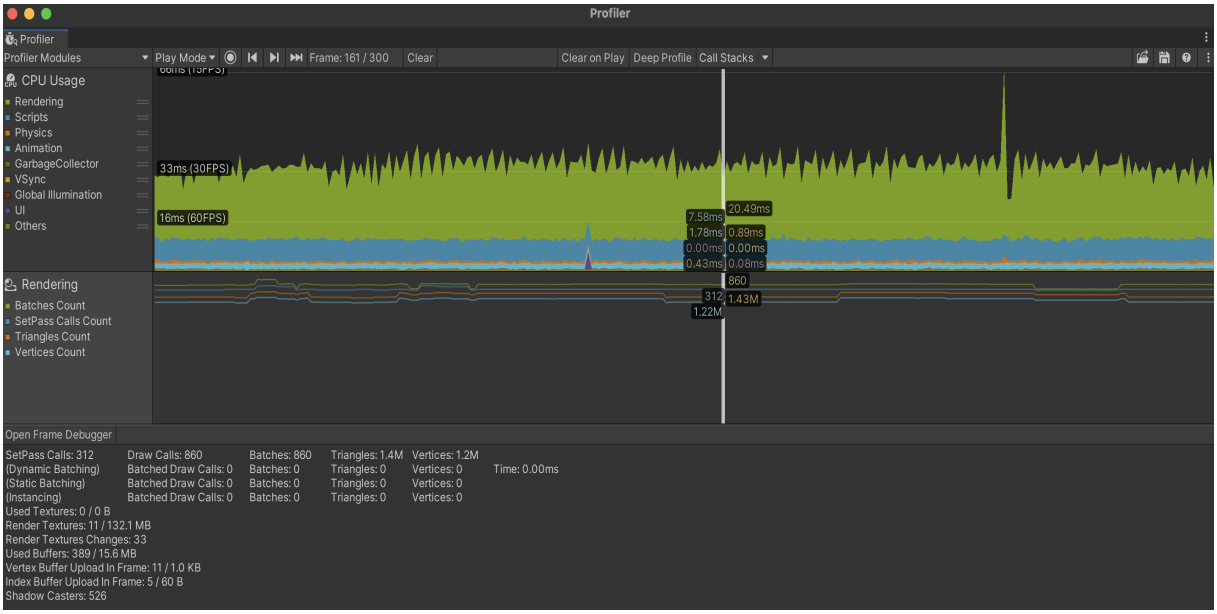


Figura 38: Profilazione generale della classe senza Deep Profiling .

Entrambi i parametri fondamentali sono ampiamente fuori dai limiti stabiliti descritti nel capitolo 4.1.1: le draw calls sono ben 860 contro le 150-250 indicate da Meta, mentre i tempi di esecuzione superano i 33ms a frame, ottenendo quindi

un frame rate inferiore a 30 fps contro il minimo di 72 indicato da Meta per un'app in VR.

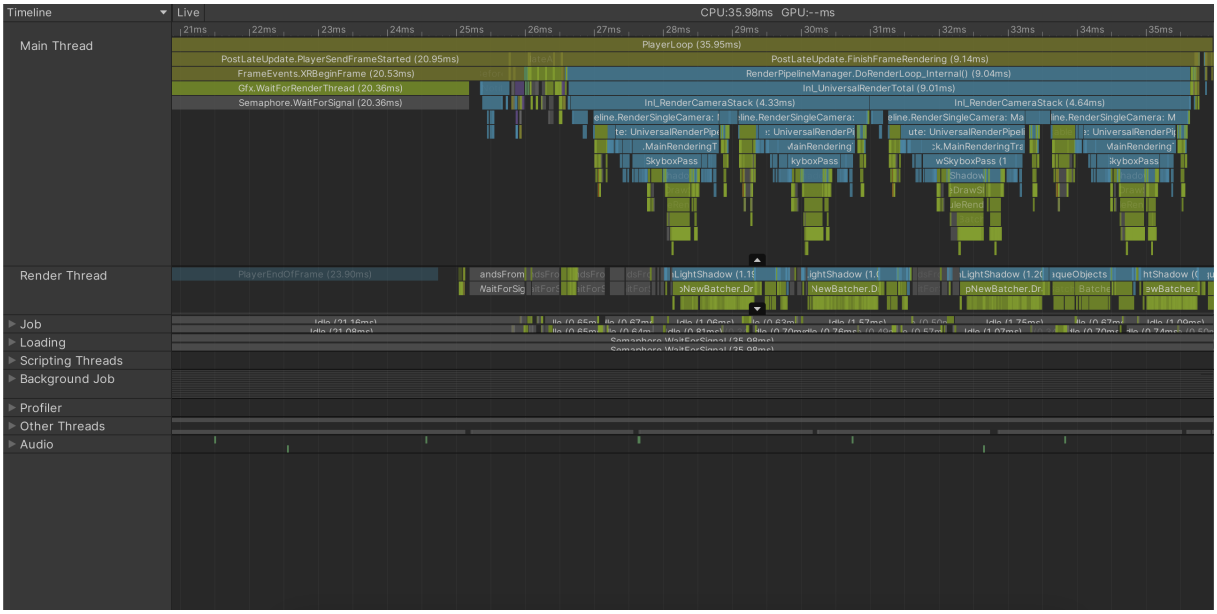
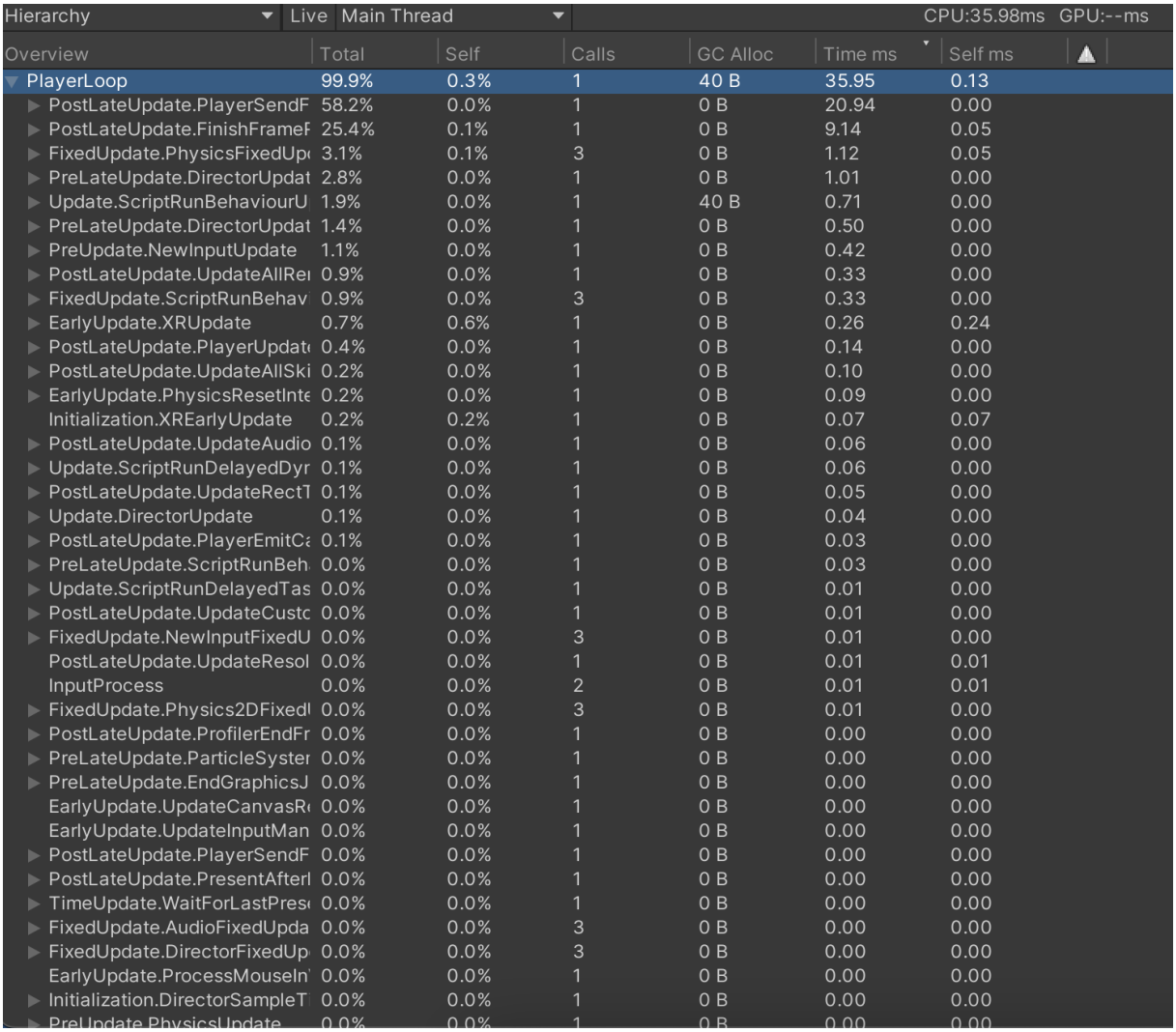


Figura 39: Profilazione generale della classe: Timeline view .



Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
PlayerLoop	99.9%	0.3%	1	40 B	35.95	0.13
PostLateUpdate.PlayerSendF	58.2%	0.0%	1	0 B	20.94	0.00
PostLateUpdate.FinishFrameF	25.4%	0.1%	1	0 B	9.14	0.05
FixedUpdate.PhysicsFixedUp	3.1%	0.1%	3	0 B	1.12	0.05
PreLateUpdate.DirectorUpdat	2.8%	0.0%	1	0 B	1.01	0.00
Update.ScriptRunBehaviourU	1.9%	0.0%	1	40 B	0.71	0.00
PreLateUpdate.DirectorUpdat	1.4%	0.0%	1	0 B	0.50	0.00
PreUpdate.NewInputUpdate	1.1%	0.0%	1	0 B	0.42	0.00
PostLateUpdate.UpdateAllRe	0.9%	0.0%	1	0 B	0.33	0.00
FixedUpdate.ScriptRunBehav	0.9%	0.0%	3	0 B	0.33	0.00
EarlyUpdate.XRUpdate	0.7%	0.6%	1	0 B	0.26	0.24
PostLateUpdate.PlayerUpdat	0.4%	0.0%	1	0 B	0.14	0.00
PostLateUpdate.UpdateAllSki	0.2%	0.0%	1	0 B	0.10	0.00
EarlyUpdate.PhysicsResetInte	0.2%	0.0%	1	0 B	0.09	0.00
Initialization.XREarlyUpdate	0.2%	0.2%	1	0 B	0.07	0.07
PostLateUpdate.UpdateAudio	0.1%	0.0%	1	0 B	0.06	0.00
Update.ScriptRunDelayedDyr	0.1%	0.0%	1	0 B	0.06	0.00
PostLateUpdate.UpdateRectT	0.1%	0.0%	1	0 B	0.05	0.00
Update.DirectorUpdate	0.1%	0.0%	1	0 B	0.04	0.00
PostLateUpdate.PlayerEmitC	0.1%	0.0%	1	0 B	0.03	0.00
PreLateUpdate.ScriptRunBeh	0.0%	0.0%	1	0 B	0.03	0.00
Update.ScriptRunDelayedTas	0.0%	0.0%	1	0 B	0.01	0.00
PostLateUpdate.UpdateCustc	0.0%	0.0%	1	0 B	0.01	0.00
FixedUpdate.NewInputFixedU	0.0%	0.0%	3	0 B	0.01	0.00
PostLateUpdate.UpdateResol	0.0%	0.0%	1	0 B	0.01	0.01
InputProcess	0.0%	0.0%	2	0 B	0.01	0.01
FixedUpdate.Physics2DFixedI	0.0%	0.0%	3	0 B	0.01	0.00
PostLateUpdate.ProfilerEndFr	0.0%	0.0%	1	0 B	0.00	0.00
PreLateUpdate.ParticleSyster	0.0%	0.0%	1	0 B	0.00	0.00
PreLateUpdate.EndGraphicsJ	0.0%	0.0%	1	0 B	0.00	0.00
EarlyUpdate.UpdateCanvasR	0.0%	0.0%	1	0 B	0.00	0.00
EarlyUpdate.UpdateInputMan	0.0%	0.0%	1	0 B	0.00	0.00
PostLateUpdate.PlayerSendF	0.0%	0.0%	1	0 B	0.00	0.00
PostLateUpdate.PresentAfterI	0.0%	0.0%	1	0 B	0.00	0.00
TimeUpdate.WaitForLastPres	0.0%	0.0%	1	0 B	0.00	0.00
FixedUpdate.AudioFixedUpda	0.0%	0.0%	3	0 B	0.00	0.00
FixedUpdate.DirectorFixedUp	0.0%	0.0%	3	0 B	0.00	0.00
EarlyUpdate.ProcessMouseIn	0.0%	0.0%	1	0 B	0.00	0.00
Initialization.DirectorSampleT	0.0%	0.0%	1	0 B	0.00	0.00
PreUpdate.PhysicsUpdate	0.0%	0.0%	1	0 B	0.00	0.00

Figura 40: Profilazione generale della classe: Hierarchy View del Main Thread .

Dei due thread più principali quello più occupato è chiaramente il Main Thread, come si vede dalla timeline in figura 39.

In particolare, analizzando i processo del Main Thread (fig. 40) notiamo come, dei 35 ms totali, 20 vengono spesi dal processocPostLateUpdate.PlayerSendFrame, marker che indica che il thread è fortemente GPU Bound e la CPU sta aspettando [**?**][1].

[1] `https://discussions.unity.com/t/what-is-postlateupdate-finishframerendering-on-profiler-and-w`
240026

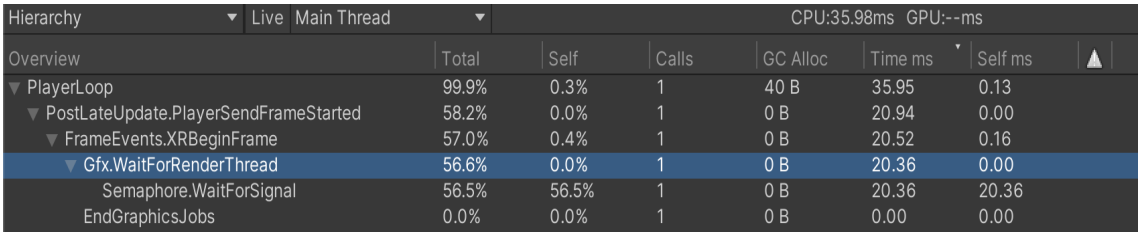


Figura 41: Analisi del marker PostLateUpdate.PlayerSendFrame .

Selezionando la voce infatti vediamo due marker, Gfx.WaitForRenderThread, al cui interno troviamo il marker Semaphore.WaitForSignal, che occupano ben 20ms, altra indicazione del fatto che il collo di bottiglia dell'app è nella parte grafica [2].

Andando ad analizzare il Render Thread però (fig. 42 notiamo che non ci sono processi molto dispendiosi ad eccezione di uno, PlayerEndOfFrame, che occupa da solo il 67% del tempo di esecuzione del Render Thread. Questo marker indica che la CPU sta aspettando la GPU [3]; addirittura il Render Thread spende 3ms aspettando comandi dal Main Thread, come notiamo dal marker Gfx.WaitForGfxCommandsFromMainThread, quindi il problema non è nei due thread bensì nella GPU.

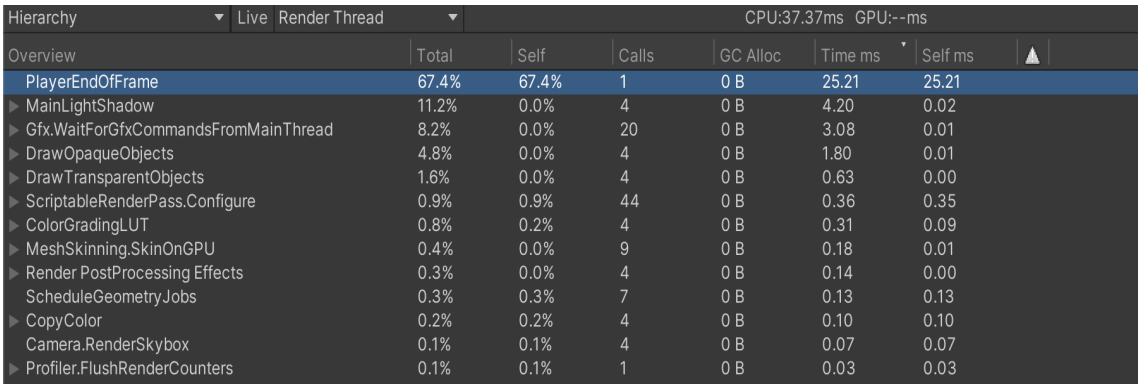


Figura 42: Analisi del Render Thread .

Anche osservando i dati del frame debugger e di Render Doc (fig. 43 e **??**) si nota che ci sono parecchi oggetti renderizzati nella scena e che i tempi di rendering della GPU sono altissimi.

[2] `https://forum.unity.com/threads/why-semaphore-waitforsignal-so-high.1021234/`

[3] `https://forum.unity.com/threads/what-is-playerendofframe.408115/`

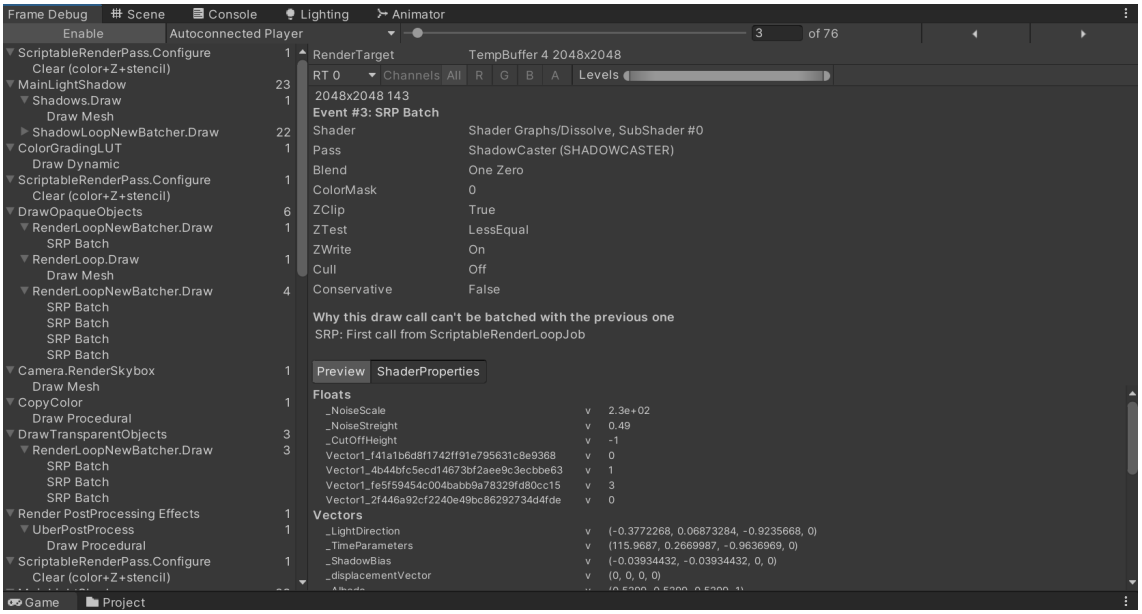


Figura 43: Analisi della classe con il Frame Debugger .

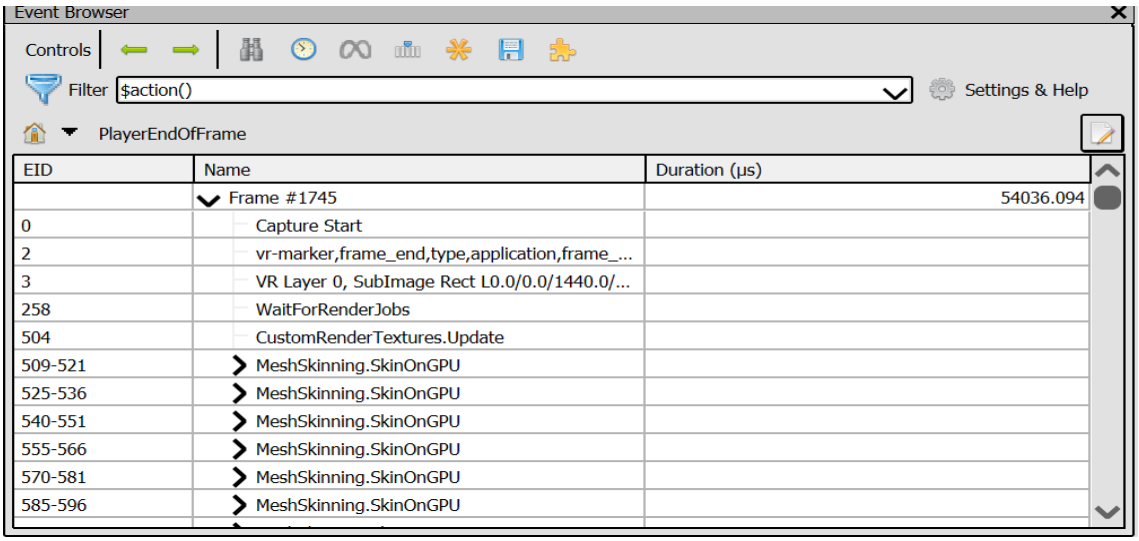


Figura 44: Analisi della classe con RenderDoc .

Anche i dati raccolti con OVRMetricTools per entrambe le scene confermano un frame rate molto basso (fig. 45 e 46).

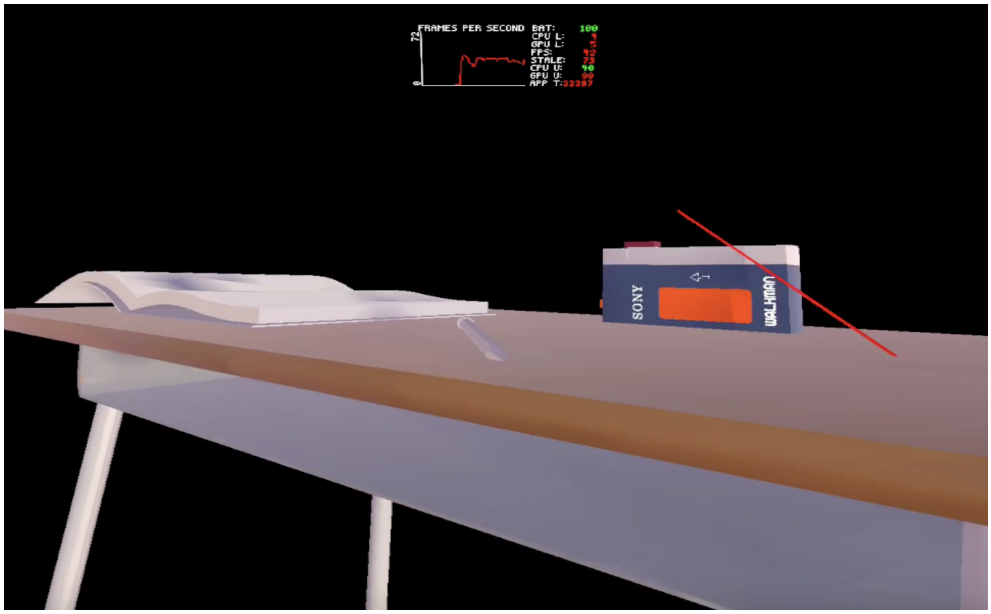


Figura 45: Analisi della scena del banco di Robin con OVRMetrics .

Figura 46: Analisi della scena della classe con OVRMetrics .

La profilazione rifatta con il deep profiling ha confermato i risultati ottenuti con le profilazioni ad alto livello senza aggiungere informazioni rilevanti; qui sotto viene

comunque mostrata la profilazione effettuata con il deep profiling per confermare quanto detto nel capitolo **??** riguardo all'overhead generato da tale profilazione.

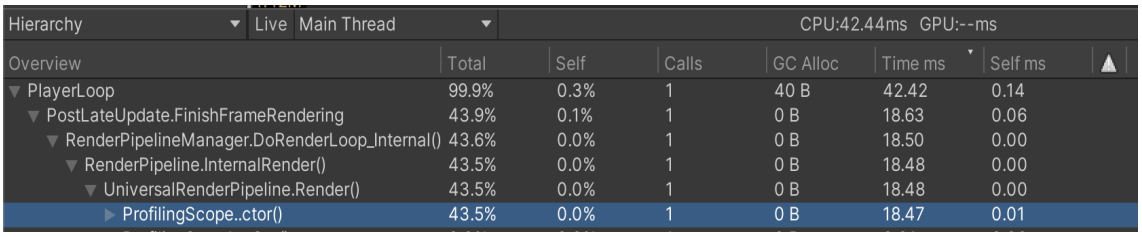


Figura 47: Analisi fatta con il Deep Profiling .

Come si può notare si hanno ben 16ms attribuiti al marker ProfilingScope.

7.3 Ottimizzazione

Qui di seguito verrà descritta l'applicazione di tutte le tecniche di ottimizzazione descritte nel capitolo 6 alla prima scena di LockedUp.

7.3.1 LOD

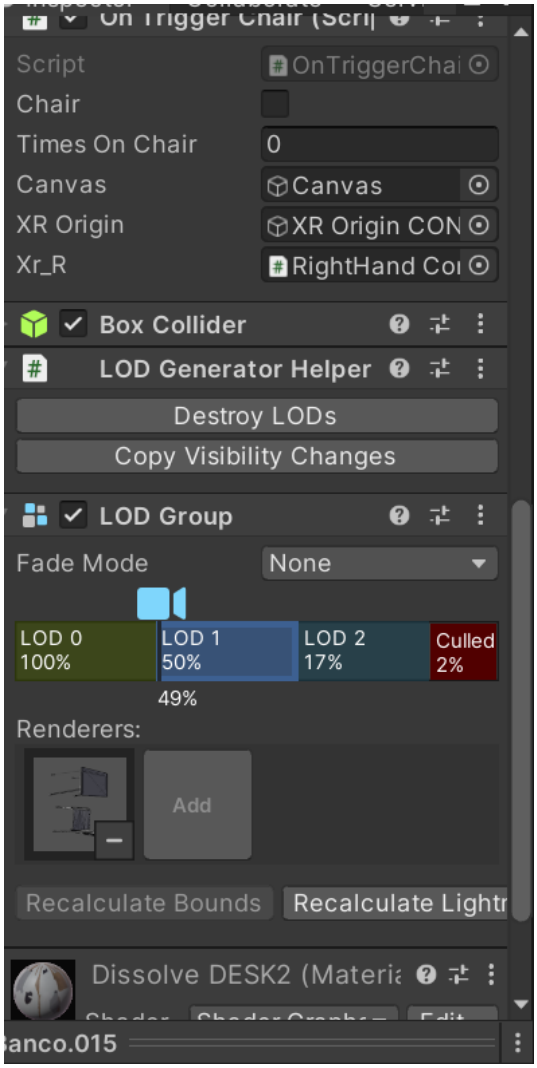


Figura 48: Impostazione del LOD sul modello del banco di Robin .

La tecnica dei LOD è stata subito scartata a causa dell'effetto di popping descritto a capitolo 6.1. Trovandoci in un ambiente piccolo infatti è molto evidente la transizione tra un modello e l'altro. In prima battuta si è provato a rendere questa distanza molto breve ma ciò creava degli artefatti molto evidenti, come mostrato nelle figure 49 e 50.

Figura 49: LOD del banco visto da lontano .

Figura 50: LOD del banco visto da vicino .

Anche regolando la distanza di transizione a valori più alti, si notavano degli artefatti poco visibili dagli screenshot ma molto visibili durante l'esperienza (fig. 51 e 52). Se si osserva attentamente si nota la differenza tra i due LOD nella seduta della sedia.

Figura 51: LOD del banco visto da lontano .

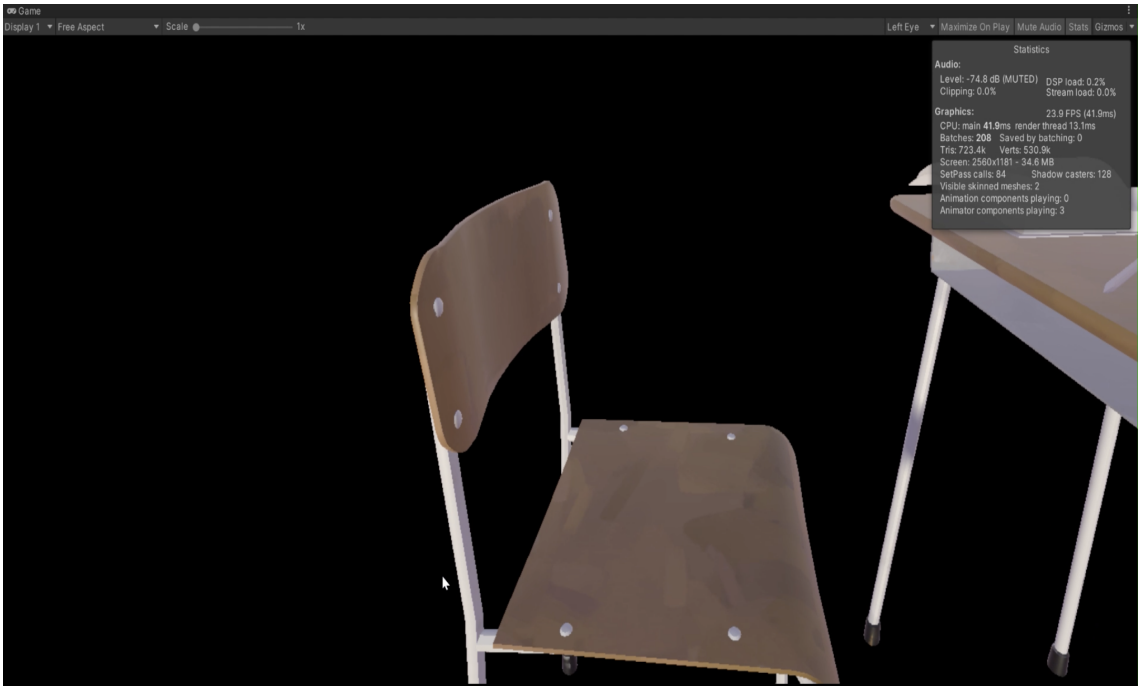


Figura 52: LOD del banco visto da vicino .

Per rendere la transizione tra LOD invisibile si sarebbe dovuta impostare la distanza di transizione a un valore tale da non essere mai raggiunta, quindi la tecnica del LOD è stata immediatamente scartata.

7.3.2 Batching e GPU Instancing

Il batching statico e dinamico e il GPU Instancing, probabilmente a causa della diversità delle keywords degli shader e in generale della diversità dei materiali usati, non hanno avuto nessun tipo di effetto sull'applicazione e sul numero di draw call, che è rimasto esattamente lo stesso, come mostrato in figura 53.

Figura 53: Effetto del Batching e del GPU Instancing .

7.3.3 Culling

Il culling ha avuto un effetto fortemente positivo sull'applicazione. Il tempo di esecuzione lato CPU è sceso sui 16-17ms (fig. 55) con dei picchi oltre i 20, una situazione nettamente migliore rispetto a quella iniziale in cui i tempi di esecuzione superavano i 33ms (fig. 37). E' leggermente aumentato il tempo speso nel marker "Gfx.WaitForGfxCommandsFromMainThread", il che potrebbe far pensare che parte del carico di lavoro si sia spostato su quel thread, mentre sono diminuiti i tempi dei marker "Gfx.WaitForRenderThread" e "PlayerEndOfFrame", fino a diventare quasi trascurabili (fig. 56 e 57)

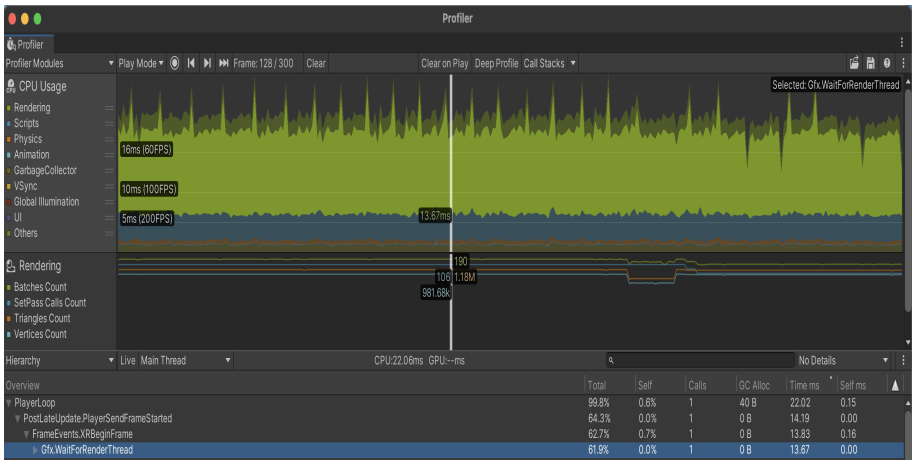


Figura 54: Effetto del culling - prima parte della scena .

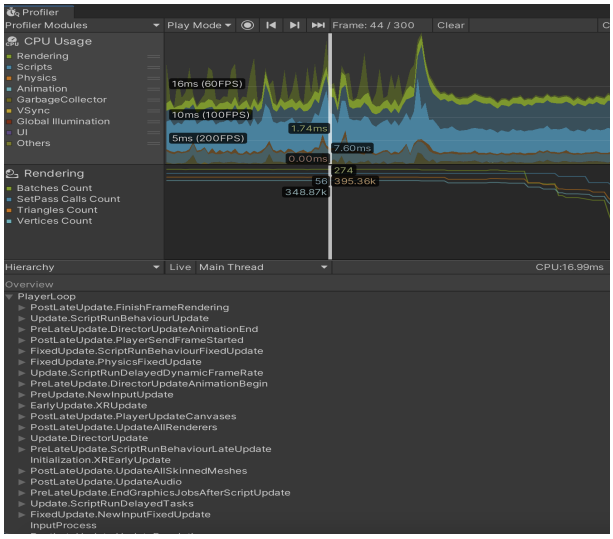


Figura 55: Effetto del culling .

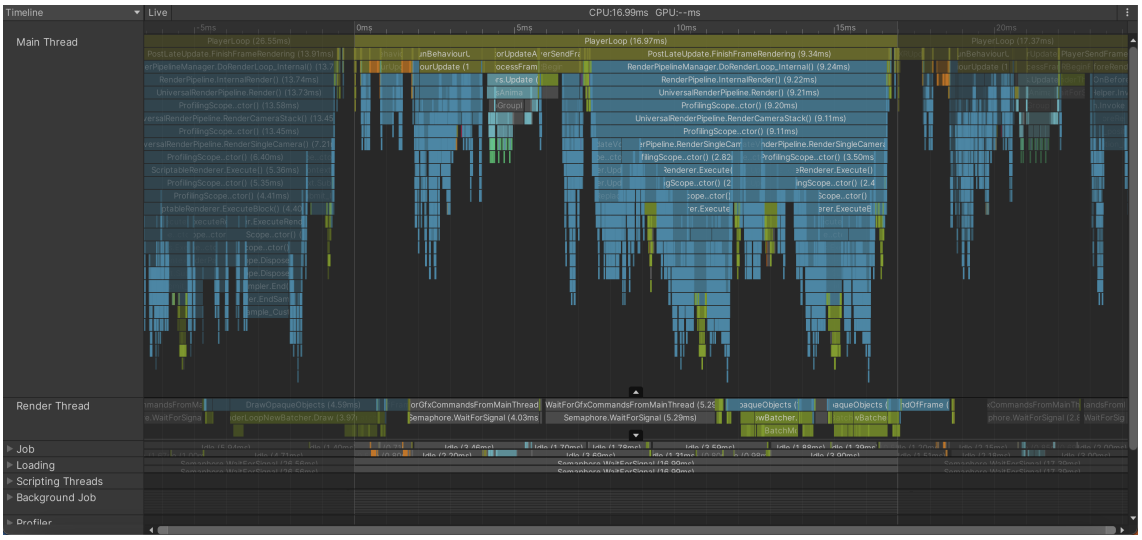


Figura 56: Effetto del culling - Timeline View .

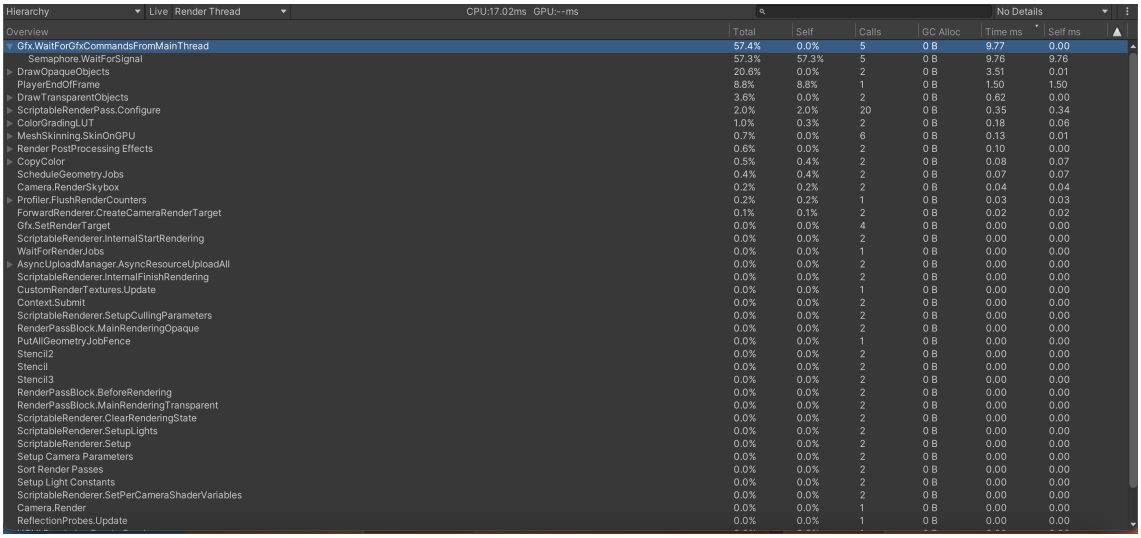


Figura 57: Effetto del culling - Render Thread .

Anche le draw calls sono notevolmente diminuite, assestandosi ora sulle 270-280 circa (fig 58, quindi già in linea con il range massimo di 200-300 draw calls suggerito da Meta.

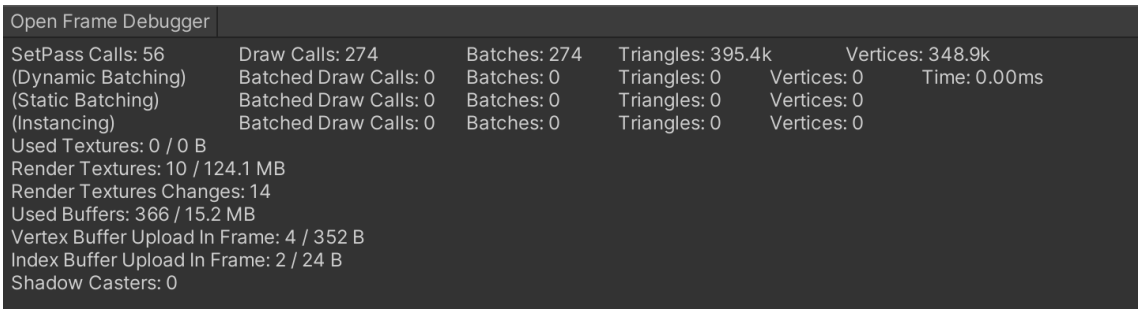


Figura 58: Effetto del culling - Draw Calls .

Qui di seguito viene invece mostrata l'analisi effettuata con RenderDoc della scena della classe (fig. 59) e del banco di Robin (fig. 60).

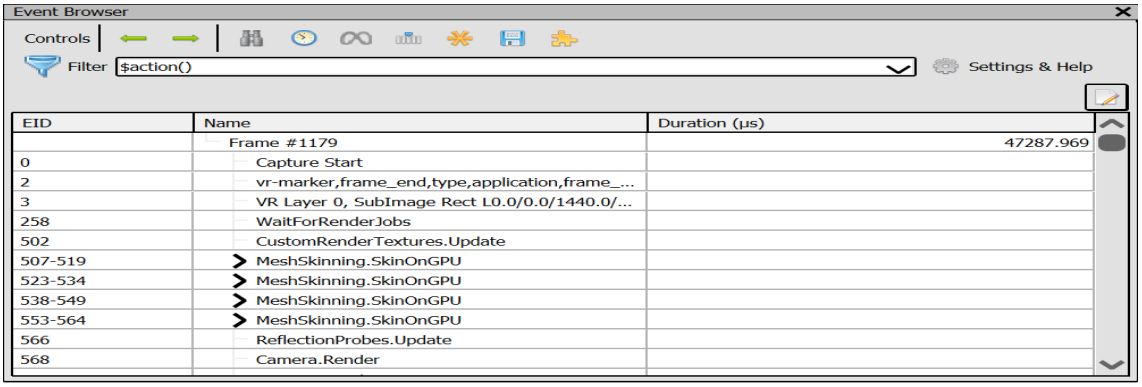


Figura 59: Analisi della classe con RenderDoc .

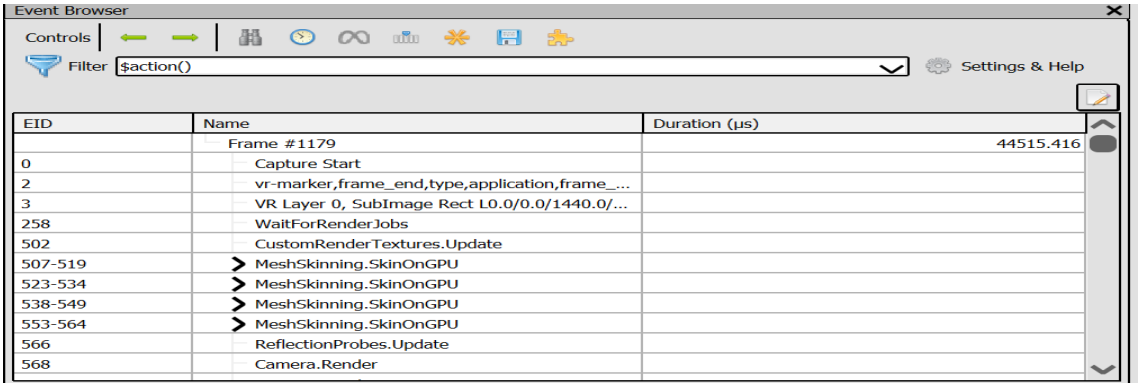


Figura 60: Analisi della scena del banco di Robin con RenderDoc .

Anche dalle metriche analizzate con OVR si nota un leggero miglioramento (fig. 61 e 62).

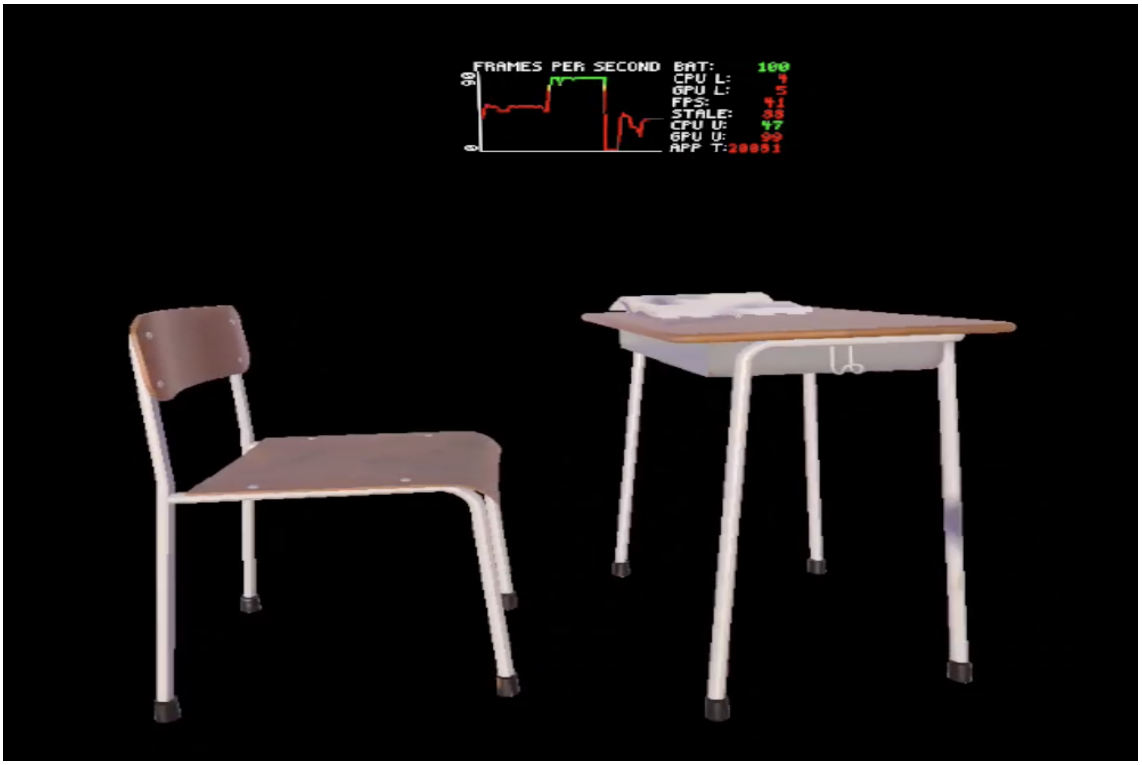


Figura 61: Analisi della scena del banco di Robin con OVRMetrics .

Figura 62: Analisi della scena della classe con OVRMetrics .

Nelle figure 63 e 64 è mostrata la scena Unity del primo atto di LockedUp con e senza culling. Il tempo di esecuzione lato GPU, pur essendo calato, è ancora molto alto.

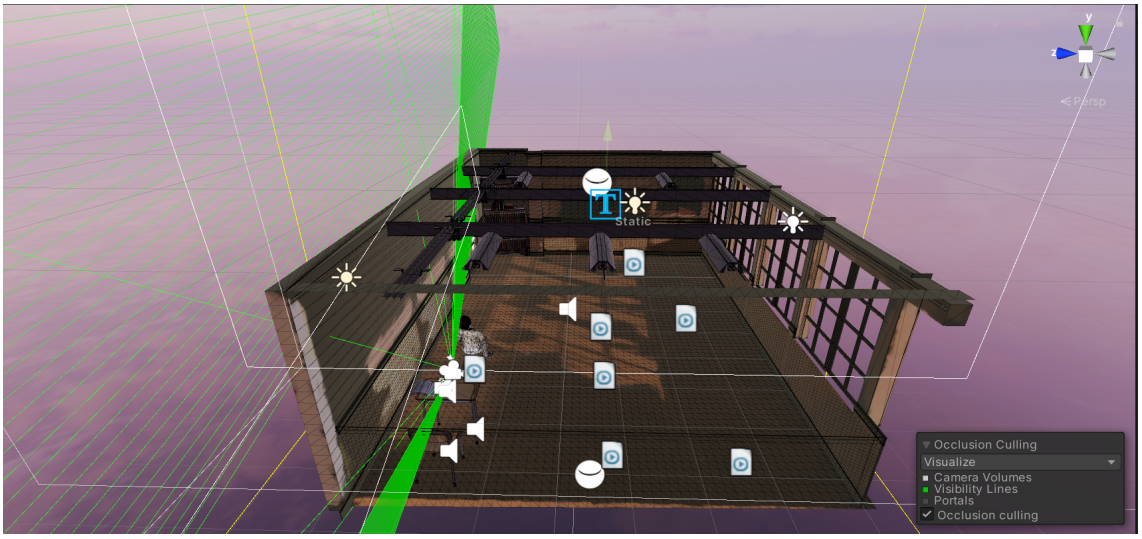


Figura 63: Scena con il culling .

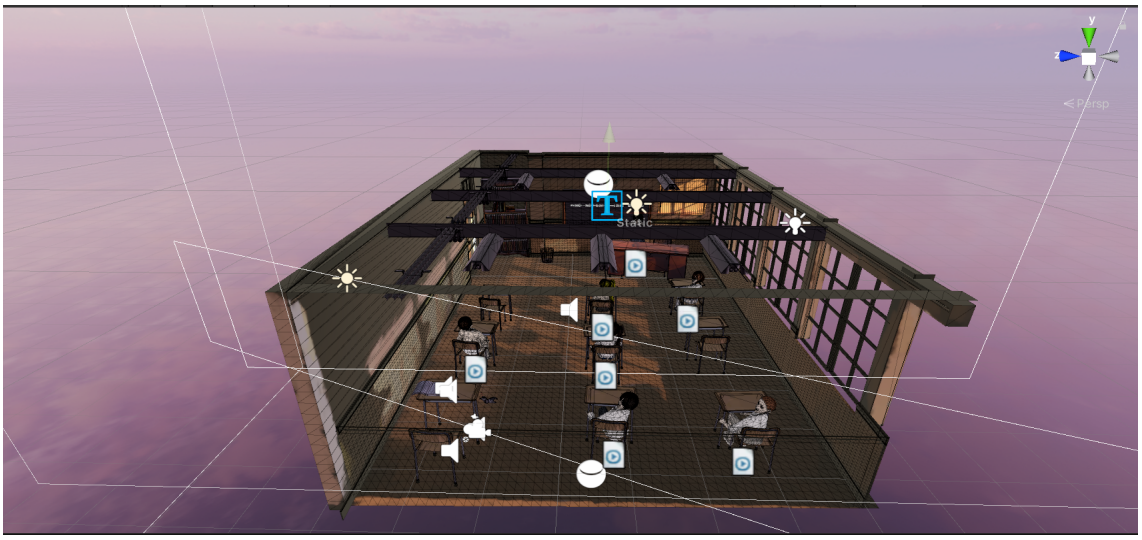


Figura 64: Scena senza il culling .

7.3.4 Light Baking

Con il baking si è ulteriormente scesi a 220 draw calls e una decina di ms a frame, anche se sono ancora presenti degli spike che portano a qualche leggero frame drop. Come si può notare però dalle figure 66 e 68 i marker "PlayerEndOfFrame" e "Gfx.WaitForRenderThread" sono ancora presenti con tempistiche non totalmente

trascurabili, seppur il tempo di esecuzione è in linea con il frame rate target di un'applicazione in VR.

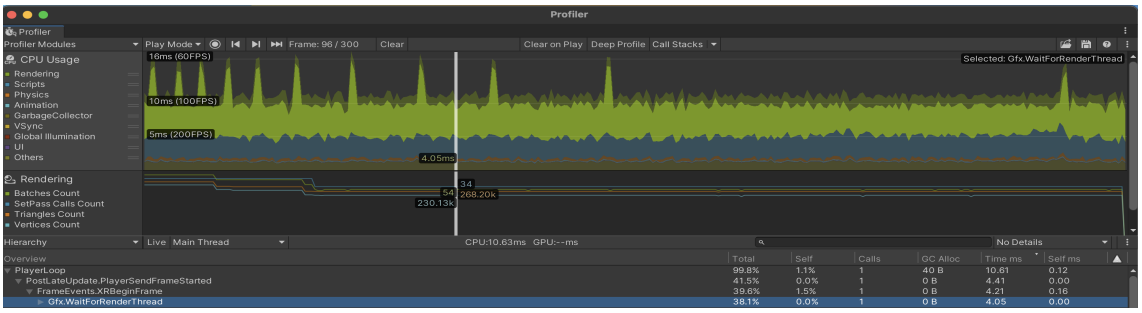


Figura 65: Effetto del baking nella scena della sedia - Main Thread .

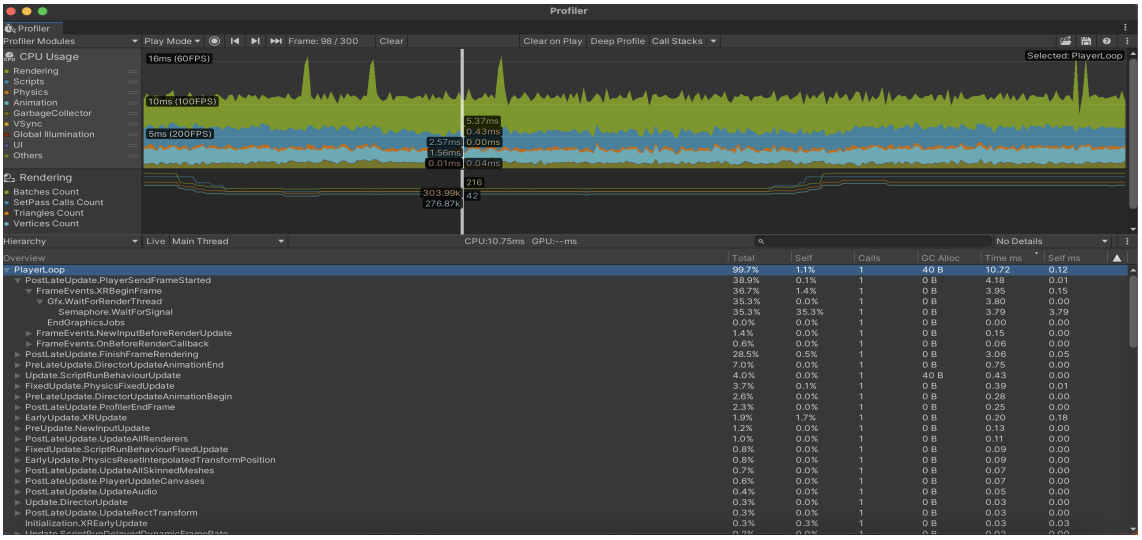


Figura 66: Effetto del baking nella scena della classe - Main Thread .

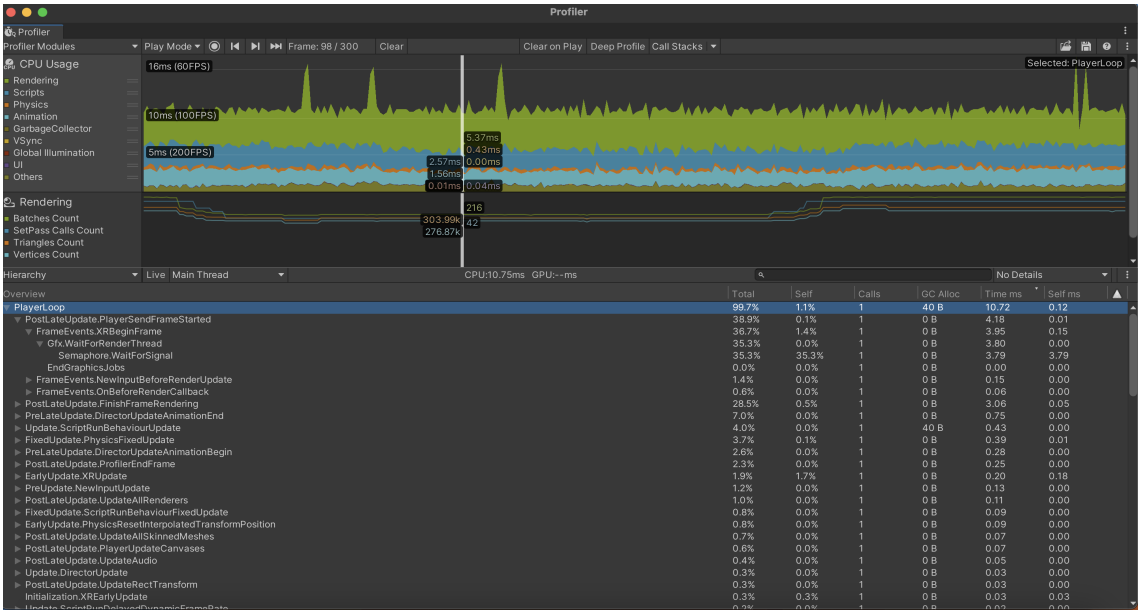


Figura 67: Effetto del baking nella scena della classe - Render Thread .

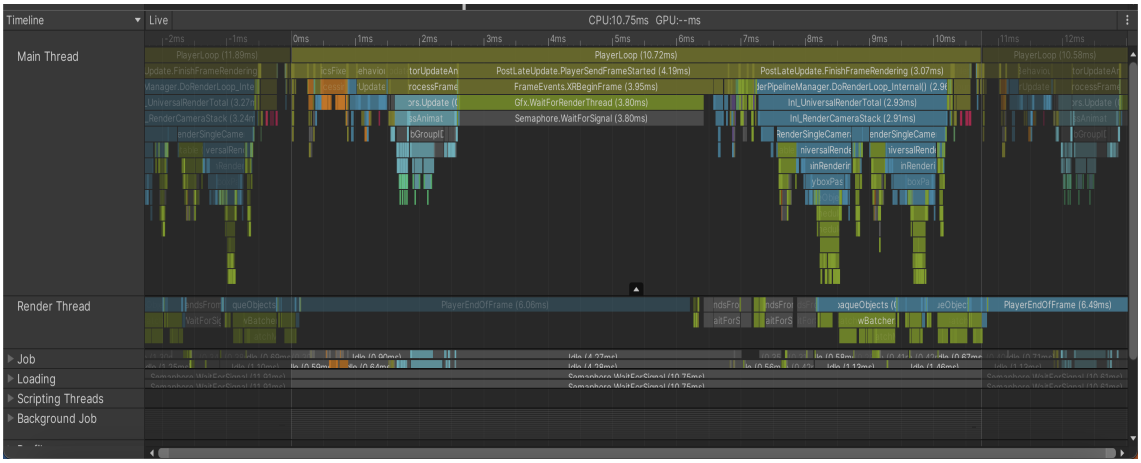


Figura 68: Effetto del baking nella scena della classe - Timeline .

Analizzando la scena tramite RenderDoc i tempi di esecuzione lato GPU sono nettamente diminuiti sia per quanto riguarda la scena della classe (fig. 69) sia per quanto riguarda la scena del banco di Robin (fig. 70).

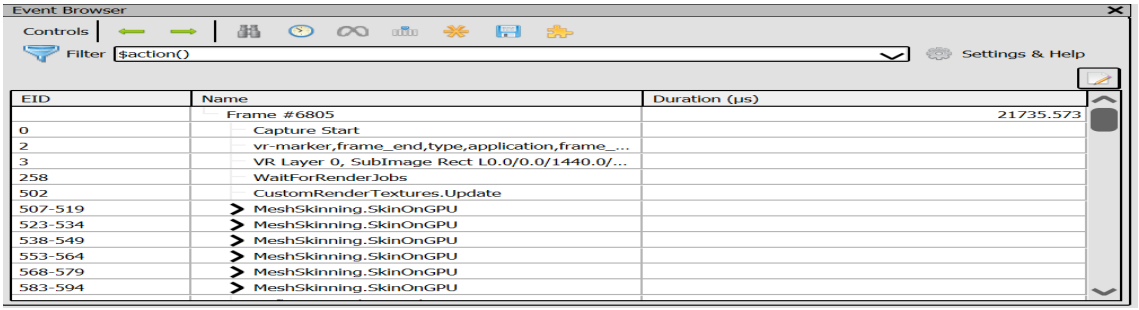


Figura 69: Effetto del baking nella scena della classe - Render Doc .

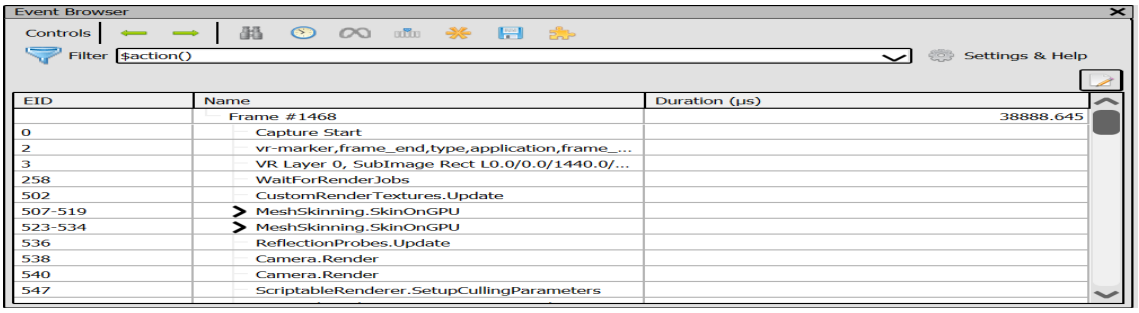


Figura 70: Effetto del baking nella scena del banco di Robin - Render Doc .

Il dato che stona in questo caso, ed è confermato anche da OVRMetricTools, è il tempo di rendering della scena del banco di Robin, che sembra molto eccessivo rispetto al tempo di rendering registrato nella scena della classe. Mentre i dati registrati con OVRMetricTools nella scena della classe (fig. 71) sono ottimi e superiori ai requisiti minimi di Meta, nella scena del banco di Robin i dati sono leggermente inferiori a quelli della classe (fig. 72) e subiscono un drop netto circa a metà della scena, prima dell'interazione con il walkmann (fig. 73).

Figura 71: Effetto del baking nella scena della classe - OVRMetricTools .

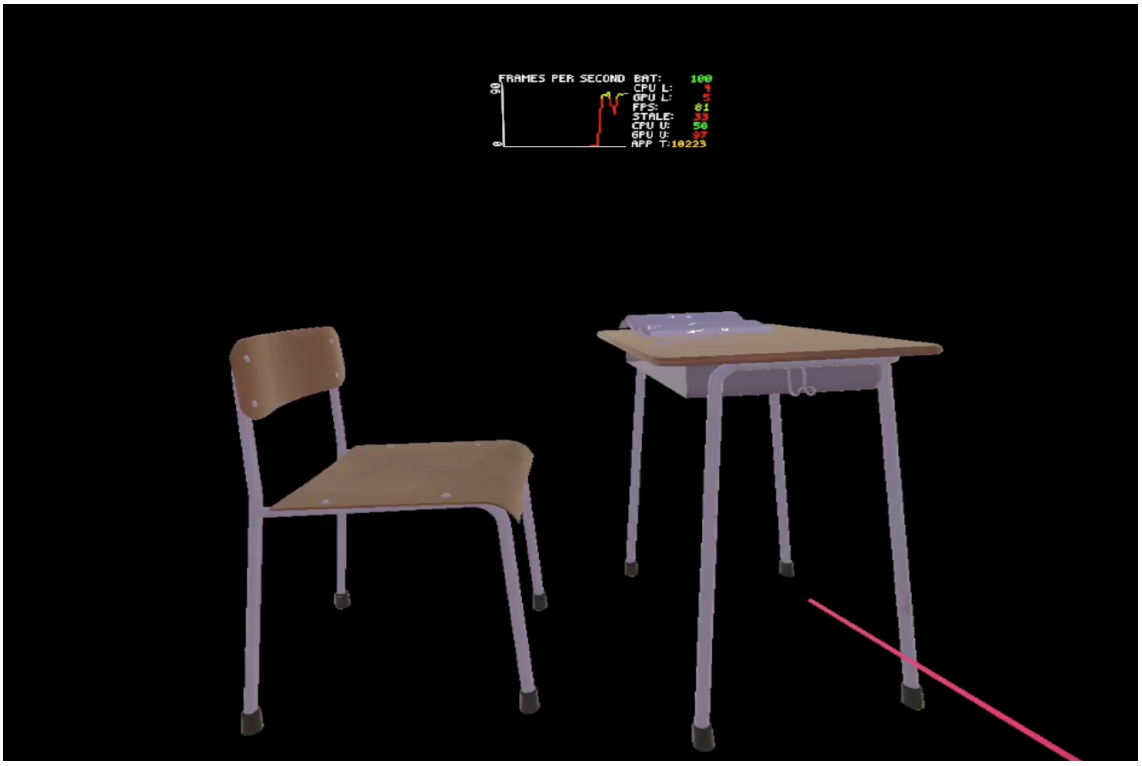


Figura 72: Effetto del baking nella scena del banco di Robin - OVRMetricTools .

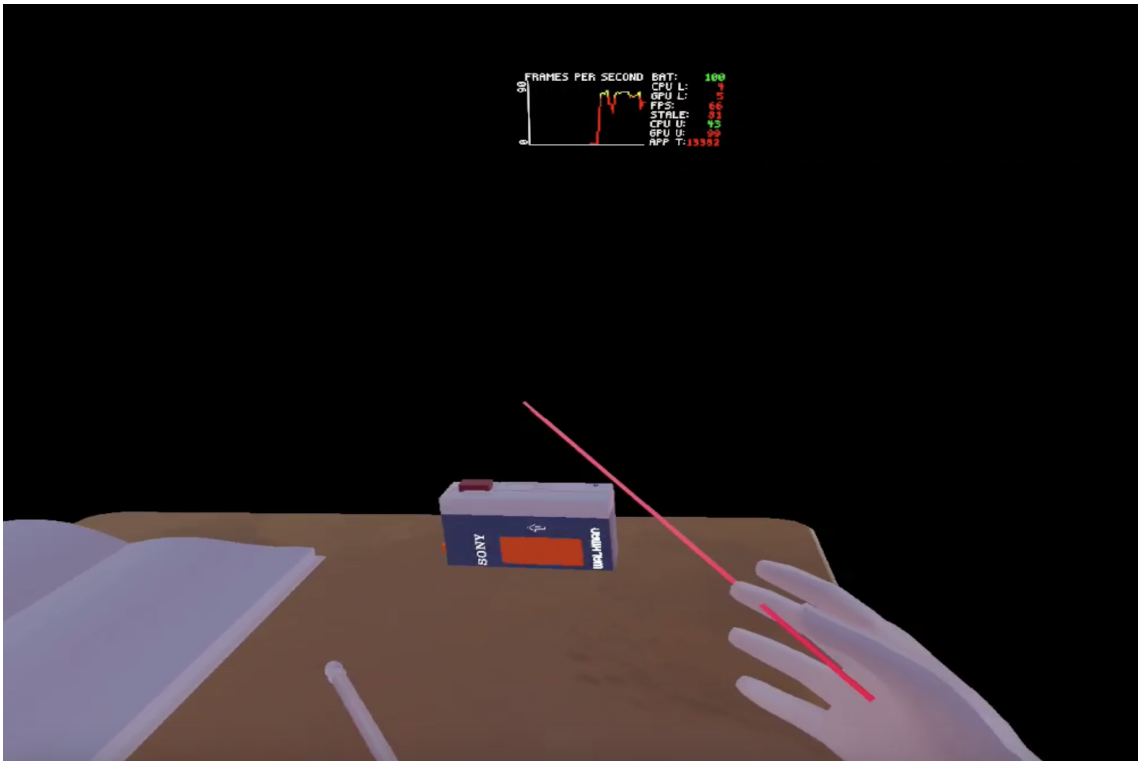


Figura 73: Drop di FPS nella scena del banco di Robin - OVRMetricTools .

7.3.5 Nuovi Modelli di Studenti e Insegnante

Per scelte stilistiche sono stati cambiati i modelli di studenti e insegnante; la scena, nei momenti in cui l'utente guarda tutta la classe, ha un leggero frame drop, comunque trascurabile, che porta il frame rate a 84-85 fps circa.

7.3.6 Ultimi accorgimenti, risultato finale e ottimizzazioni future

Per quanto riguarda il frame drop presente nella scena iniziale, quella con il banco di Robin, questo era causato dal fatto che nella scena erano attive, senza essere viste, le pareti con lo shader dell'attacco di panico. Una volta disattivati i relativi GameObject, non c'è stato nessun frame drop e la scena si è stabilizzata sui 90 fps. Per l'attacco di panico, lo shader dovrà essere sostituito da delle texture per essere reso più leggero e gestibile dalla GPU dell'Oculus Quest 2.

Capitolo 8

LockedUp, Atto 2

8.1 Introduzione

Nel secondo atto di LockedUp l'utente si trova nella camera di Robin; le luci sono inizialmente spente (74) e si accendono poco dopo (fig. 75).

Figura 74: LockedUp, atto 2, stanza con le luci spente .

Figura 75: LockedUp, atto 2, stanza con le luci accese .

L'utente può interagire con vari oggetti presenti nella stanza, come il suo orsacchiotto Dooda, o giocare con la PSP, unico punto di contatto rimasto tra Robin e i suoi amici (fig. 76). Ogni volta che si interagisce con un oggetto, si scopre qualcosa della vita di Robin. Avvicinandosi alla finestra e tirando la cordicella, l'utente assisterà, dalla stanza, agli amici di Robin che stanno giocando a basket senza di lui (fig. 77).

Figura 76: LockedUp, atto 2, interazione con la Playstation .

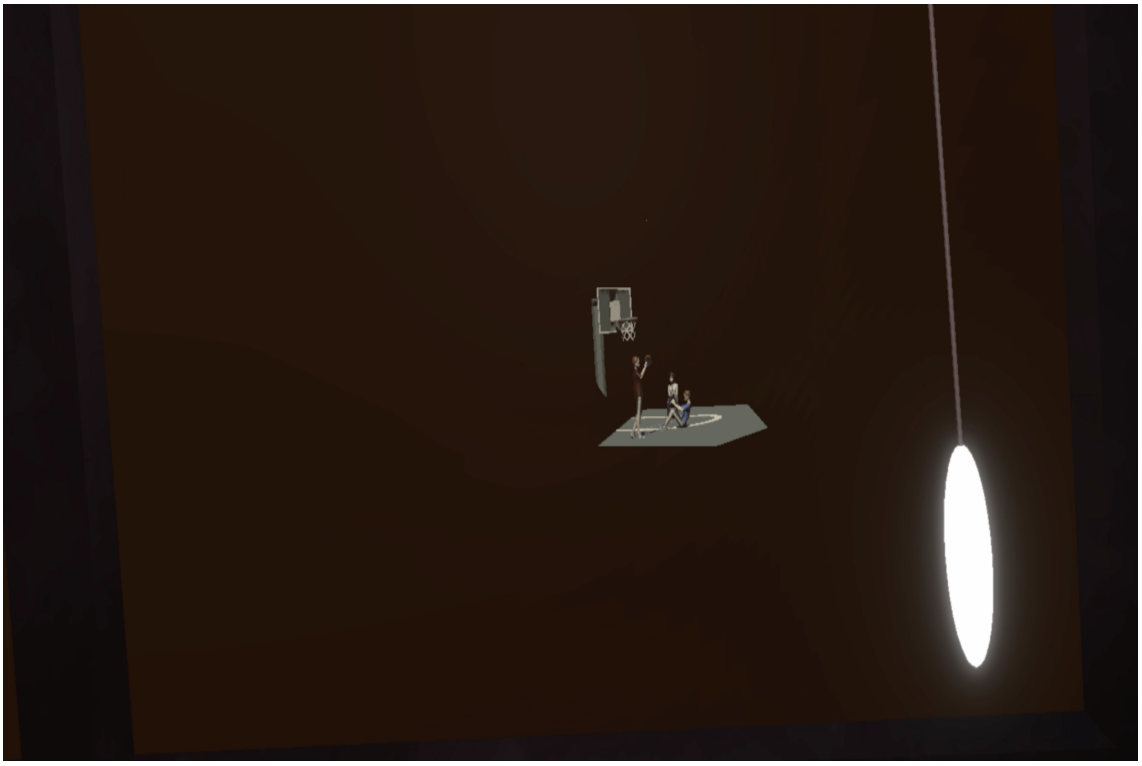


Figura 77: LockedUp, atto 2, amici di Robin che giocano a Basket .

Durante l'esplorazione, il passaggio del tempo è dato dalla comparsa nella stanza di libri, compiti, scatole e piatti sporchi. L'ultimo oggetto con cui si interagirà sarà il telefono: Robin riceverà un invito da un suo amico per giocare fuori ma non risponderà. Successivamente avrà un altro attacco di panico, questa volta con la comparsa, nell'ambiente, di frasi prounciate da Robin stesso relative alla situazione che sta vivendo (fig. 78).

Figura 78: LockedUp, atto 2, attacco di panico .

8.2 Profilazione

Il secondo atto di LockedUp si è presentato molto più fluido del primo (cap. 6.7) ancor prima di effettuare delle ottimizzazioni.

A inizio scena, prima dell'accensione delle luci, i tempi di processing sia della CPU che della GPU sono quasi in linea con il target minimo di 72 fps; tuttavia, sono ancora presenti, con valori di tempo non indifferenti, i marker PlayerEndOfFrame e Gfx.WaitForRenderThread (fig. 79, 80 e 81).

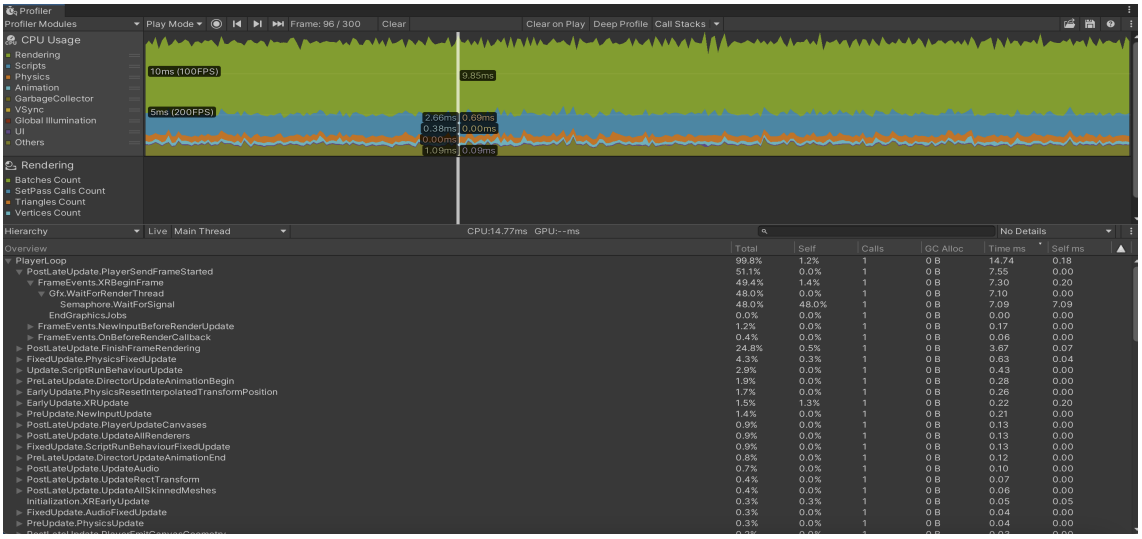


Figura 79: Atto 2, Main Thread .

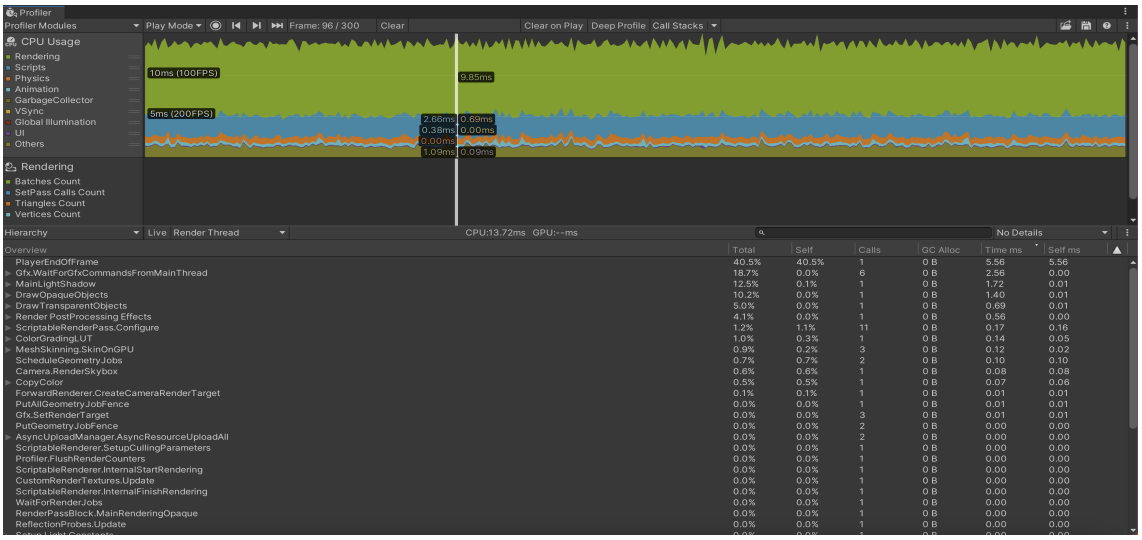


Figura 80: Atto 2, Render Thread senza ottimizzazioni .

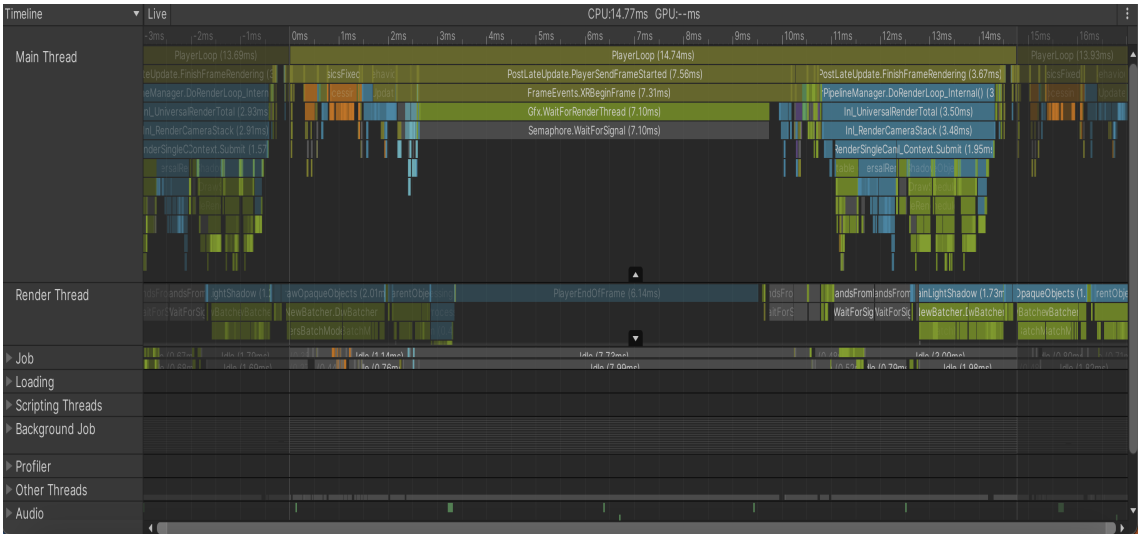


Figura 81: Atto 2, Timeline senza ottimizzazioni .

I valori registrati da Render Doc segnalano invece una situazione estremamente GPU-Bound; il tempo totale di processing della GPU è decisamente oltre al limite quando la camera ha ancora le luci spente (fig. 82) e aumenta leggermente quando le luci sono acccese (fig. 83).

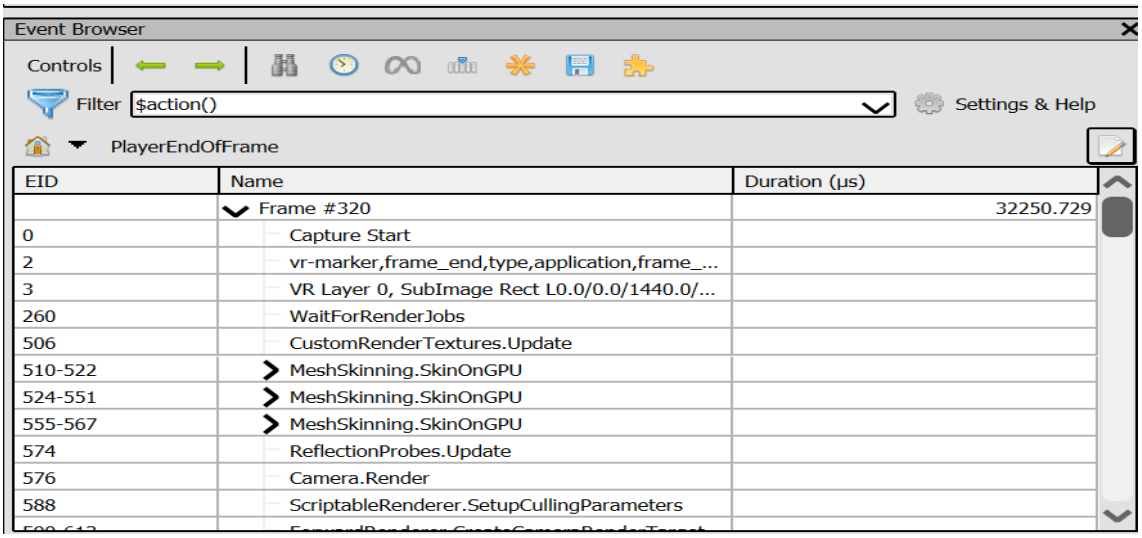
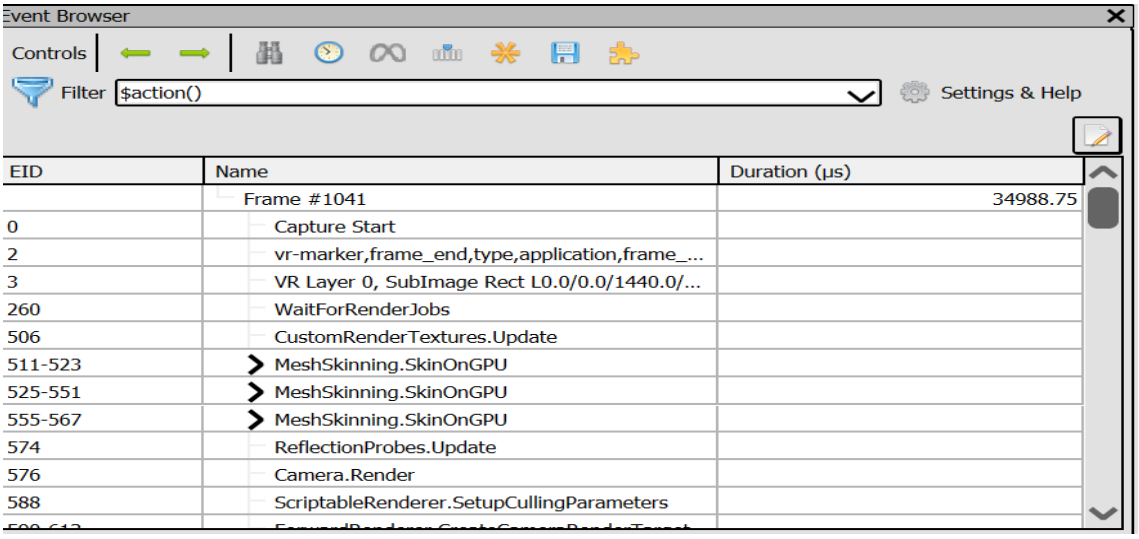


Figura 82: Atto 2, dati di Render Doc con le luci spente .



EID	Name	Duration (µs)
	Frame #1041	34988.75
0	Capture Start	
2	vr-marker,frame_end,type,application,frame_...	
3	VR Layer 0, SubImage Rect L0.0/0.0/1440.0/...	
260	WaitForRenderJobs	
506	CustomRenderTextures.Update	
511-523	MeshSkinning.SkinOnGPU	
525-551	MeshSkinning.SkinOnGPU	
555-567	MeshSkinning.SkinOnGPU	
574	ReflectionProbes.Update	
576	Camera.Render	
588	ScriptableRenderer.SetupCullingParameters	

Figura 83: Atto 2, dati di Render Doc con le luci accese .

Qui sotto vengono illustrati i risultati della profilazione nei tre momenti in cui si hanno i maggiori cali di frame per secondo: quando si ha l'effetto di passaggio della luce sui muri per far capire che sta passando il tempo (fig. 84 e 85, quando compaiono le scatole e i compiti (fig. 86, 87, 88) e quando si verifica l'attacco di panico finale (fig. 89 e 90).

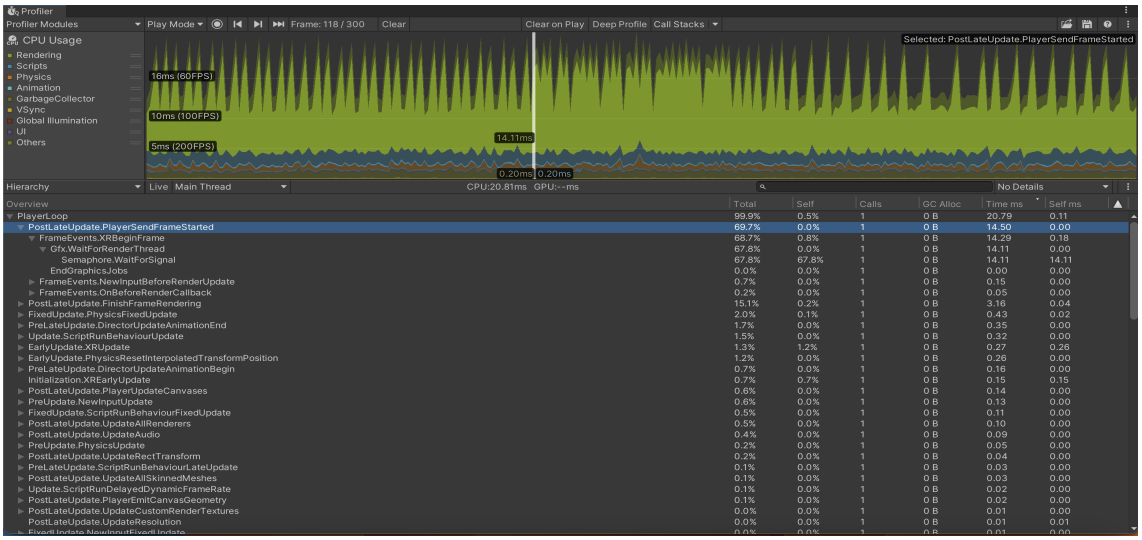


Figura 84: Atto 2, passaggio della luce, Main Thread .

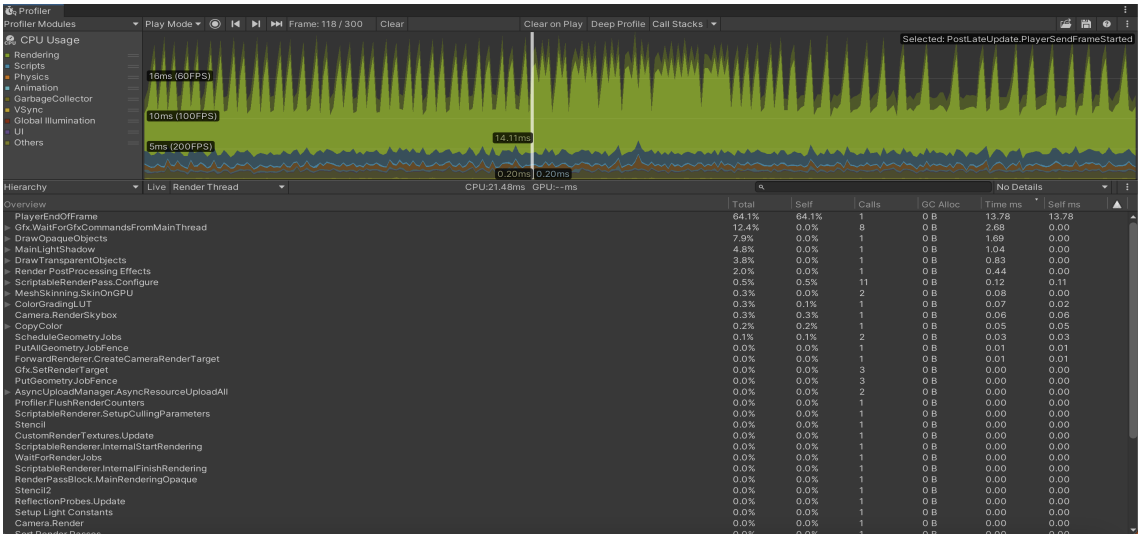


Figura 85: Atto 2, passaggio della luce, Render Thread .

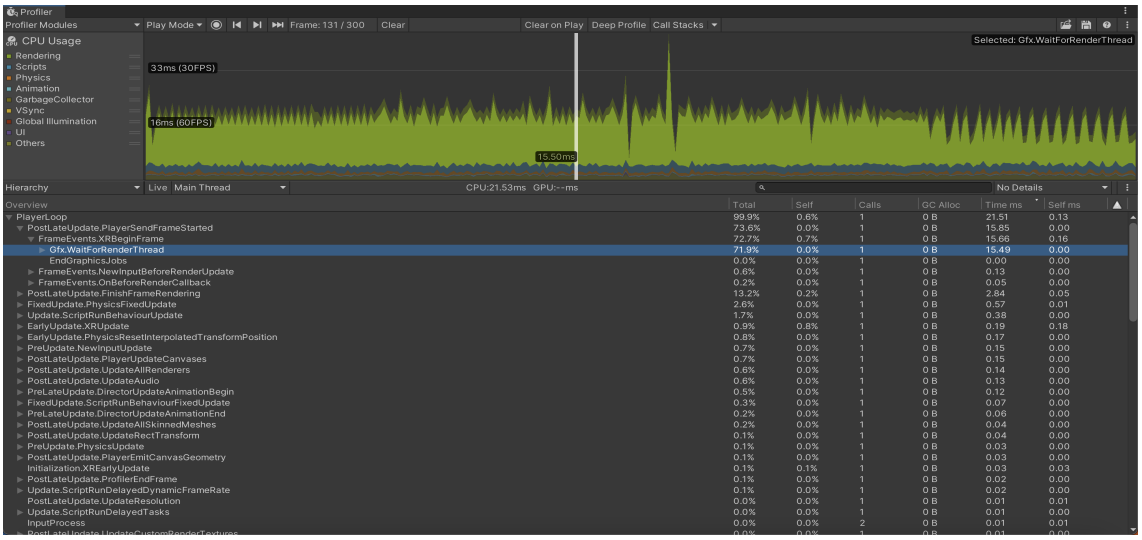


Figura 86: Atto 2, comparsa delle scatole, Main Thread .

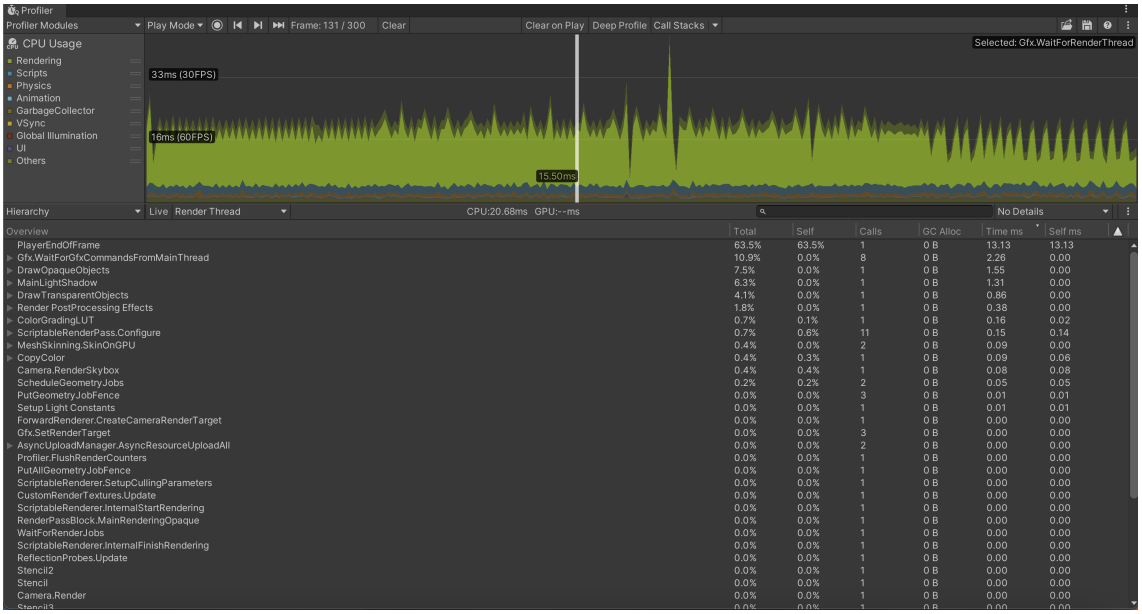


Figura 87: Atto 2, comparsa delle scatole, Render Thread .

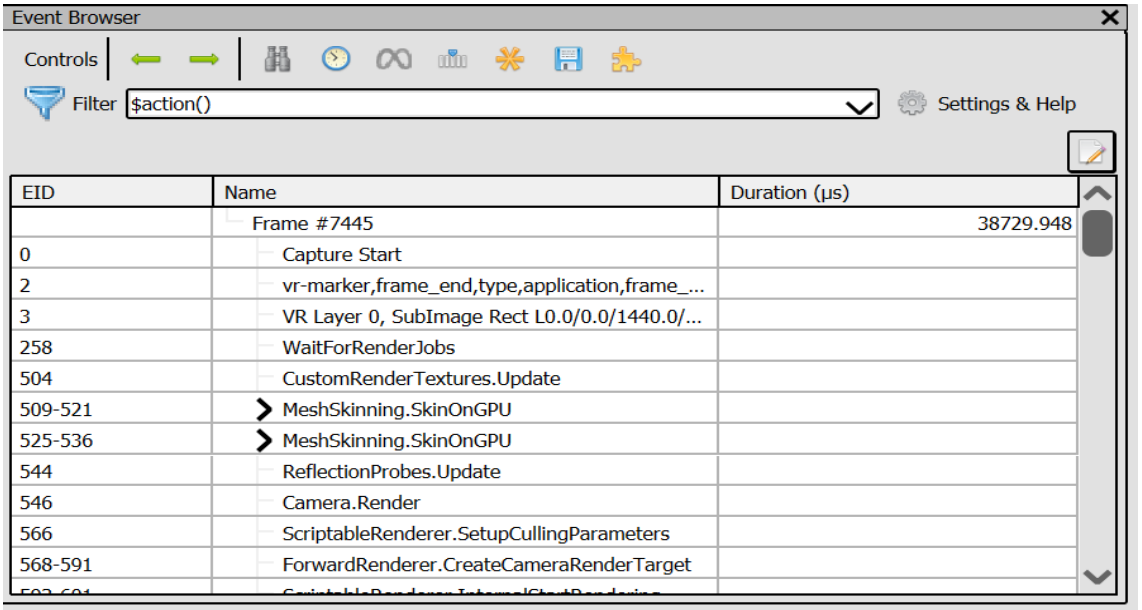


Figura 88: Atto 2, comparsa delle scatole, Render Doc .

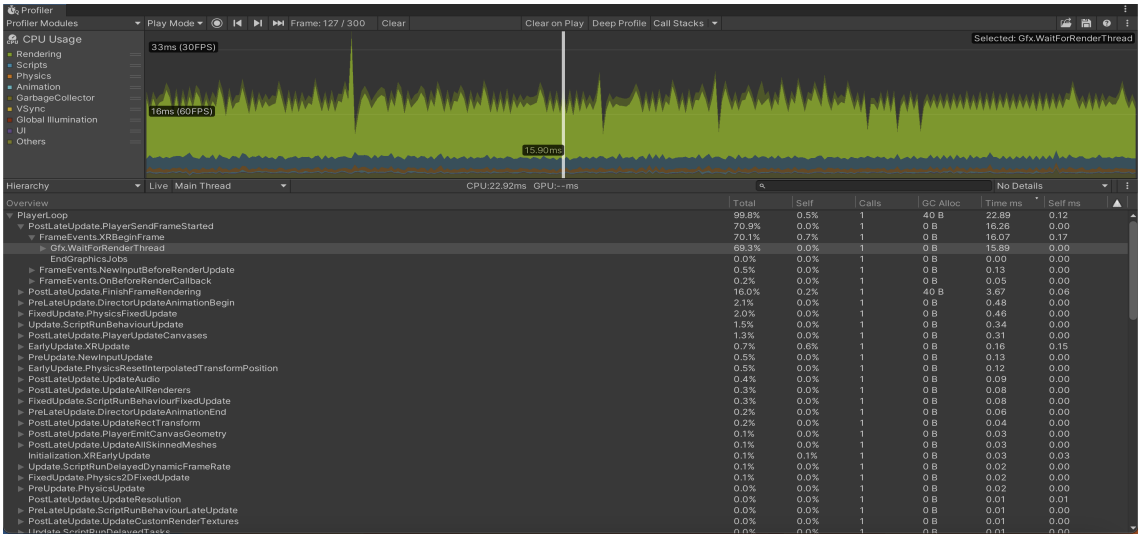


Figura 89: Atto 2, attacco di panico, Main Thread .

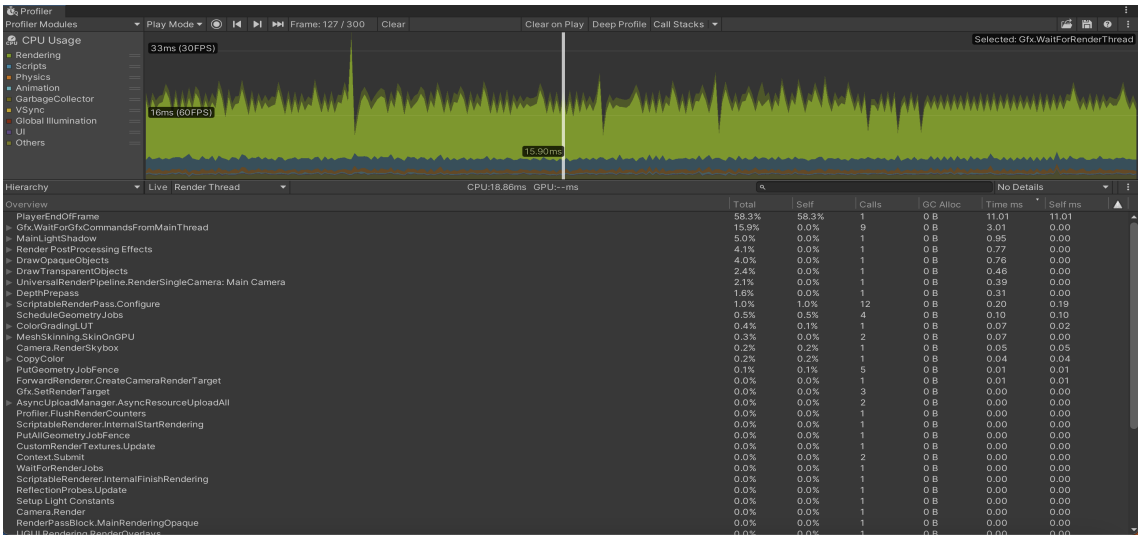


Figura 90: Atto 2, attacco di panico, Render Thread .

La scena, prima di essere ottimizzata, è stata profilata utilizzando anche OVR-MetricTools in tre momenti: a luci spente (fig. 91), a luci accese (fig. 92) e quando si guarda al punto in cui più avanti compariranno le scatole e i compiti (fig. 93). Già solo guardando il punto in cui compariranno scatole e compiti senza che questi siano effettivamente comparsi si ha un calo di frame, segno che i GameObject e i relativi shader sono in realtà già attivi.

Figura 91: Atto 2, profilazione con OVR, luci spente .

Figura 92: Atto 2, profilazione con OVR, luci accese .

Figura 93: Atto 2, profilazione con OVR, punto di comparsa dei compiti e delle scatole .

8.3 Ottimizzazione

Qui di seguito verrà descritta l'applicazione di tutte le tecniche di ottimizzazione descritte nel capitolo 6 alla seconda scena di LockedUp.

8.3.1 LOD

Visti i problemi affrontati con i LOD nell'ottimizzazione dell'atto 1 (cap. 7.3.1), e dato che anche in questo caso l'ambiente è molto piccolo, l'adozione di LOD è stata subito scartata.

8.3.2 Batching e GPU Instancing

Il batching della scena è stato effettuato sia utilizzando lo Scriptable Render Pipeline Bathcer sia senza, e successivamente sono stati confrontati i risultati per

decidere in quale delle due configurazioni si ottenessero i risultati migliori. Prima di addentrarci nell'analisi dei risultati quindi, è bene parlare brevemente dello Scriptable Render Pipeline Bathcer [1].

SRP Batcher

Lo Scriptable Render Pipeline Batcher, o SRP, è un'ottimizzazione delle draw call utilizzata da Unity. La modalità normale per ridurre le draw calls è di metterle insieme e ridurle: quello che fa SRP invece è ridurre il cambio di stati nella pipeline di render tra una draw call e l'altra, combinando i comandi **bind** e **draw**, e diminuendo quindi i setup della GPU tra una draw call e l'altra. Ogni sequenza di questi comandi è chiamata SRP Batch, e, per ottenere risultati ottimali, un SRP Batch deve avere quanti più bind e draw possibile; è bene quindi avere minor varianti possibili di uno shader. Usando SRP, il materiale di un gameObject è memorizzato nella GPU: se non cambia, non c'è bisogno che la CPU lo prenda e lo carichi nuovamente nel buffer della GPU. Utilizzando SRP Batcher, non è possibile utilizzare il GPU Instancing. In figura (fig 94) sono illustrate le differenze nella pipeline di rendering con e senza SRP Batcher.

[1] `https://docs.unity3d.com/Manual/SRPBatcher.html`

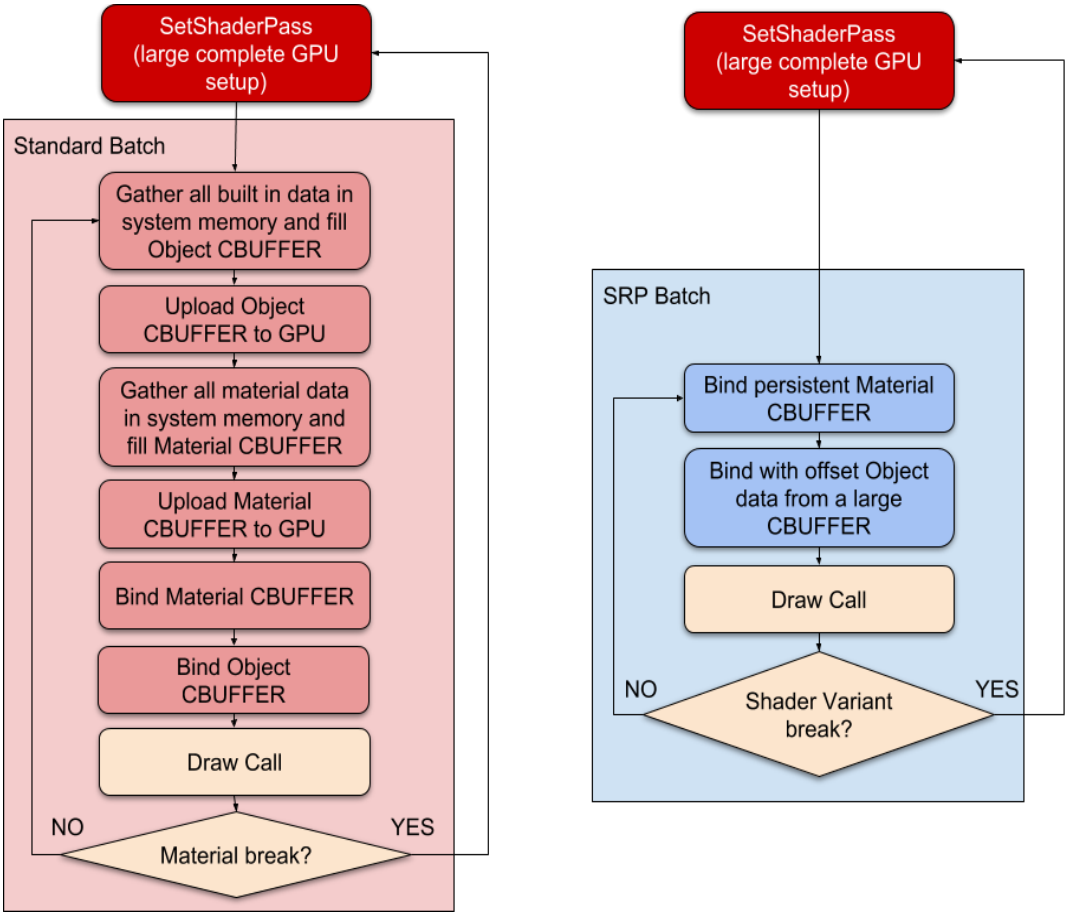


Figura 94: Differenze nella pipeline di rendering con e senza SRP Batcher .

Se si utilizza questo tipo di batcher analizzando la scena con il Frame Debugger si noterà la voce "SRP Batch", come vedremo in seguito.

Batching con SRP Batcher

Il punto più critico di tutta la scena, per quanto riguarda le draw calls, è sicuramente la comparsa delle mesh dei compiti e delle scatole, visto che sono presenti moltissime copie degli stessi (fig. 95), quindi per eseguire le profilazioni è stato preso come riferimento il momento in cui compaiono le mesh.

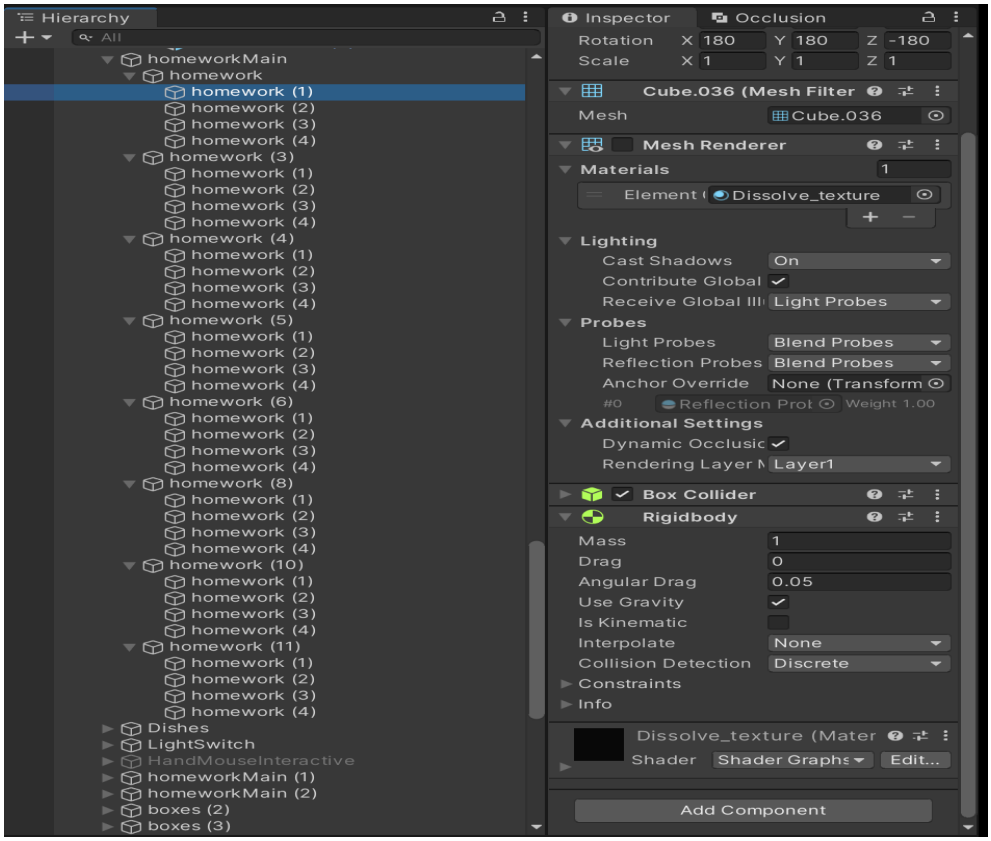


Figura 95: Atto 2, mesh dei compiti .

I tempi di processing del Main Thread e del Render Thread sono sui 17 e sugli 11 ms, quindi ancora sopra la soglia prestabilita (fig. 96 e 97).

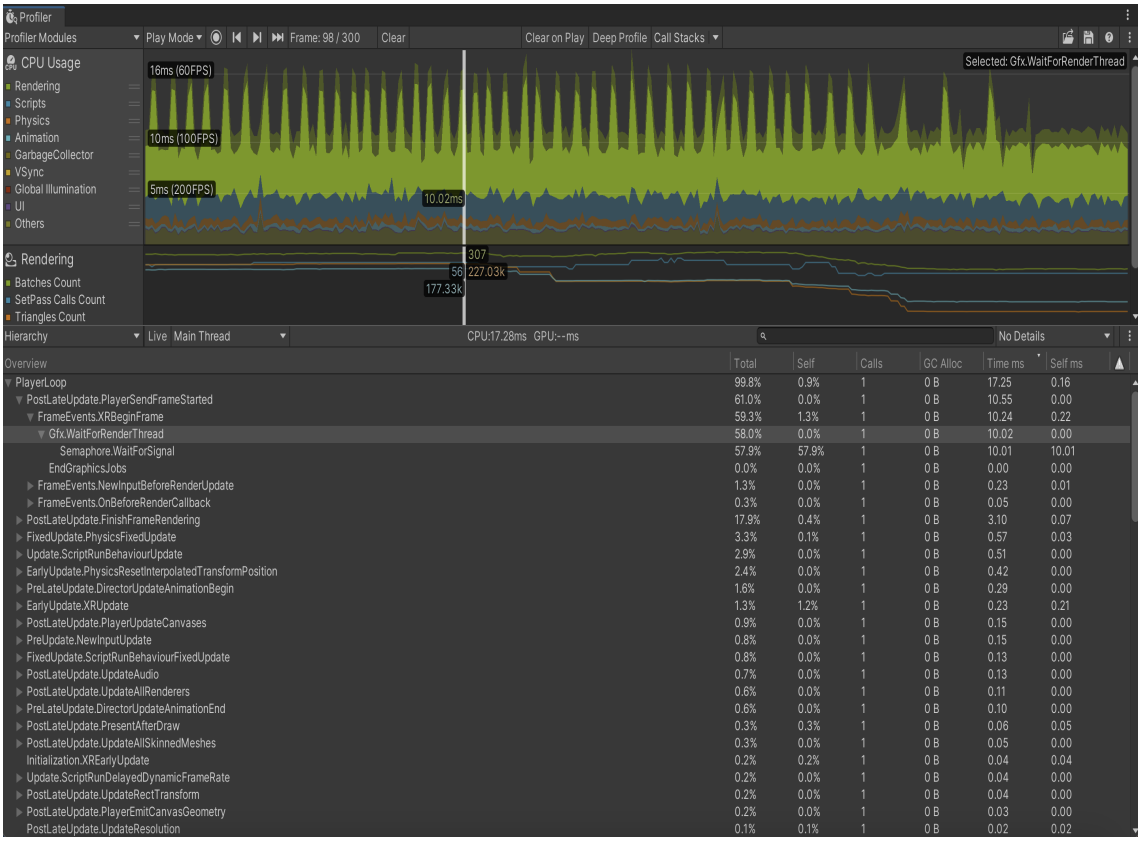


Figura 96: Atto 2, batching con SRP Batcher, Main Thread .

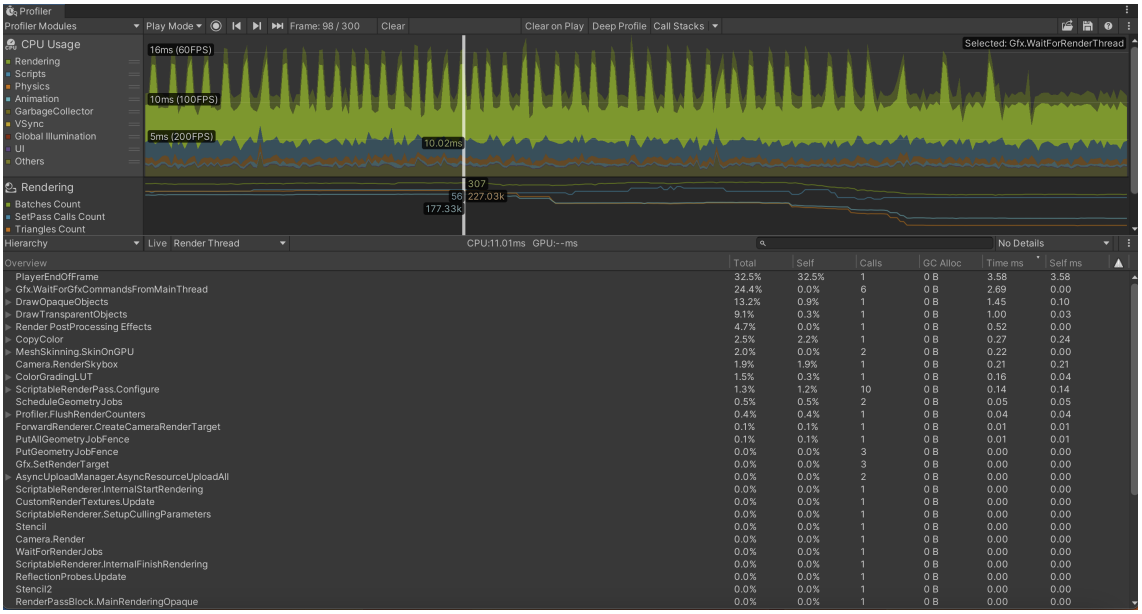


Figura 97: Atto 2, batching con SRP Batcher, Render Thread .

Le draw calls si attestano sul valore di 360-370 circa (fig. 98); dato che è in uso SRP batcher, i dati relativi allo static e al dynamic batching sono a 0 (fig. 98).

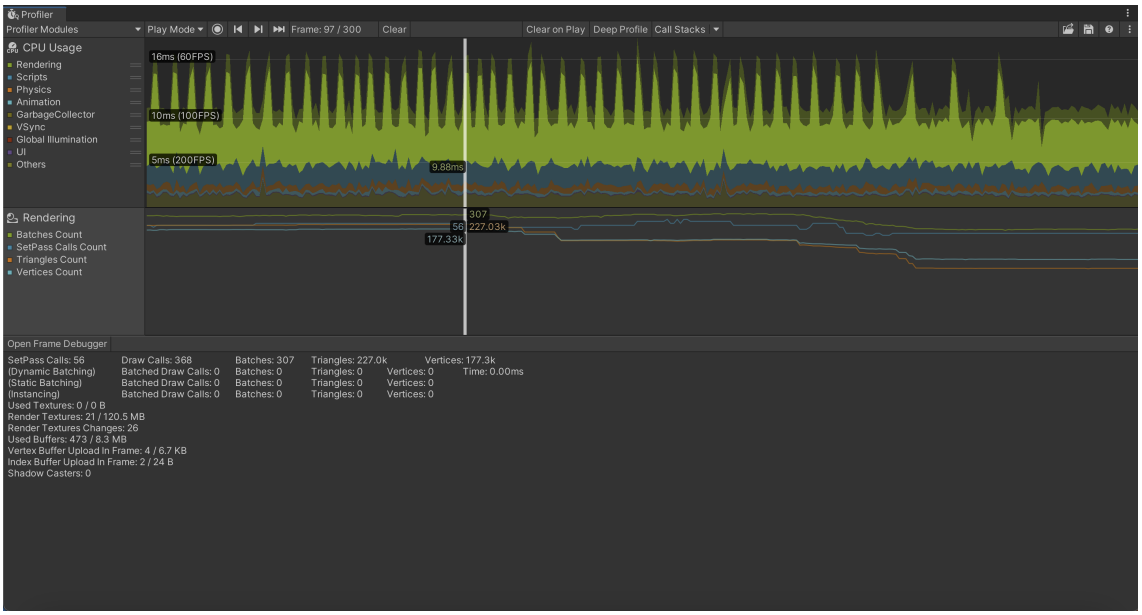


Figura 98: Atto 2, batching con SRP Batcher, dati di Render .

Per confermare che l'SRP Batcher stesse funzionando, la scena è stata analizzata utilizzando il frame debugger (fig. 99).

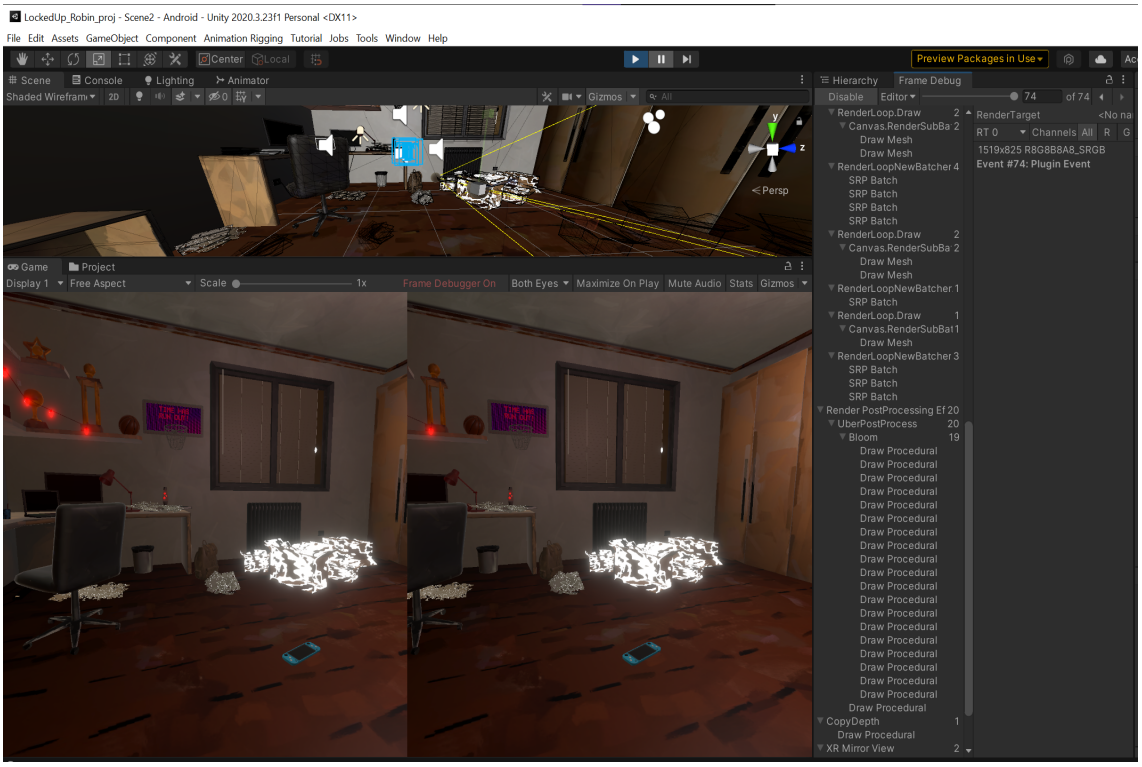


Figura 99: Atto 2, batching con SRP Batcher, frame debugger .

La voce "SRP Batch" è presente, quindi il batching con SRP Batcher sta funzionando.

I dati di RenderDoc comunque (fig. 100) evidenziano che le scatole e i compiti appesantiscono notevolmente la GPU.

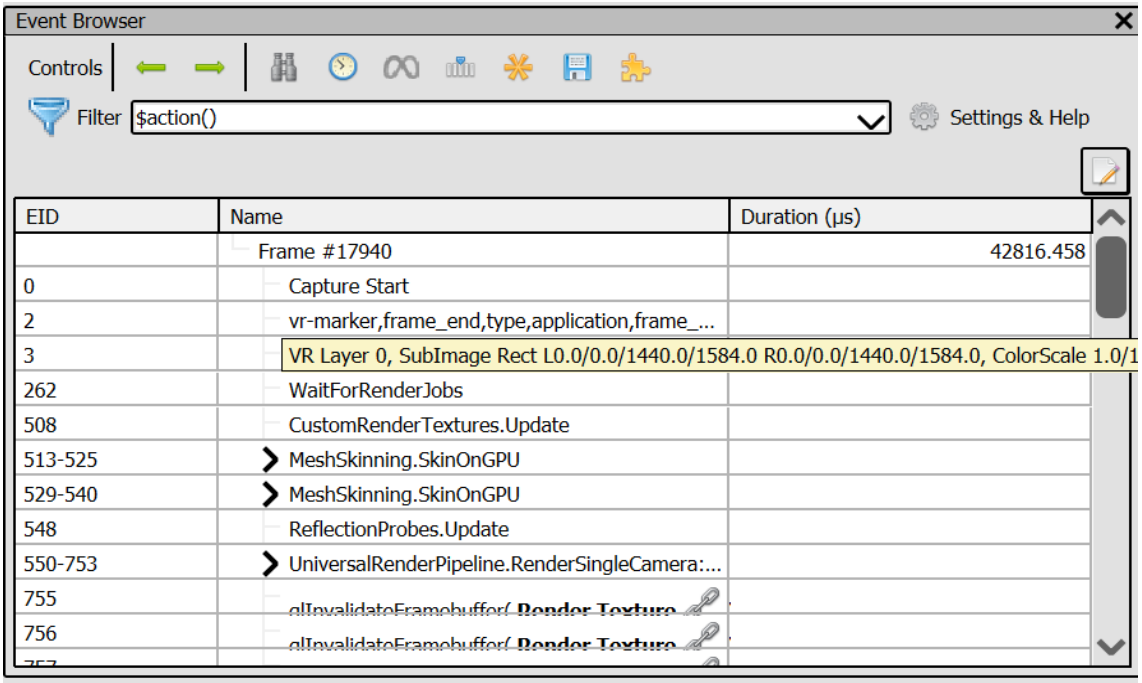


Figura 100: Atto 2, batching con SRP Batcher, Render Doc .

Batching senza SRP Batcher

I tempi di processing della CPU senza utilizzare SRP Batcher rimangono pressochè uguali, ovvero sui 10-11 ms con degli spike improvvisi a 17 ms (fig. 101).

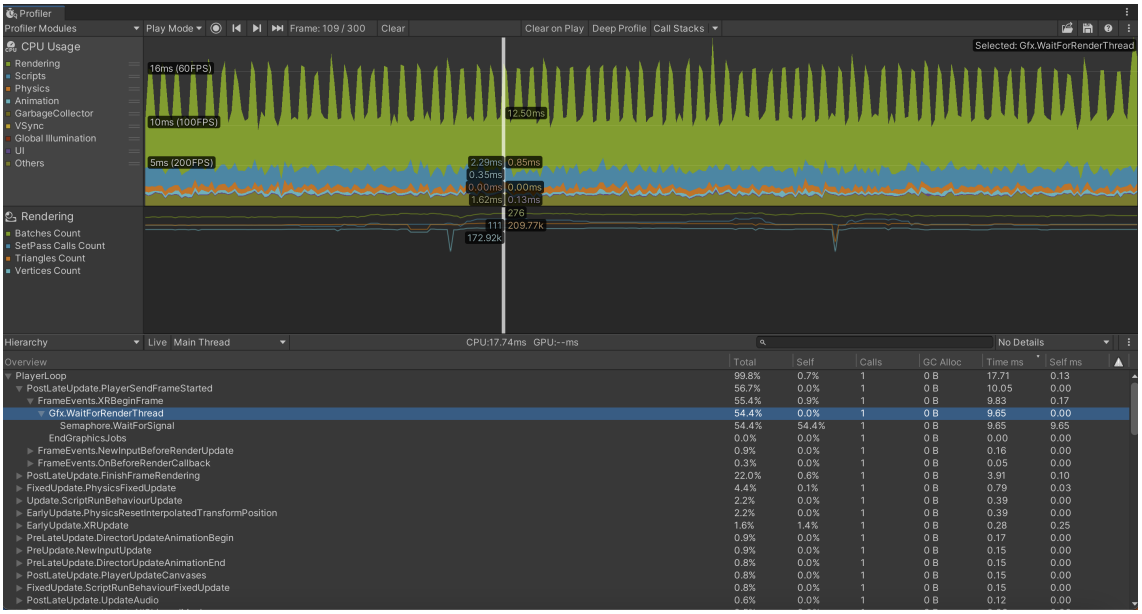


Figura 101: Atto 2, batching senza SRP Batcher, Main Thread .

Per quanto riguarda le draw calls, c'è stato un leggero calo, di circa 30 draw calls, e molte di queste sono state ottimizzate (fig. 102).

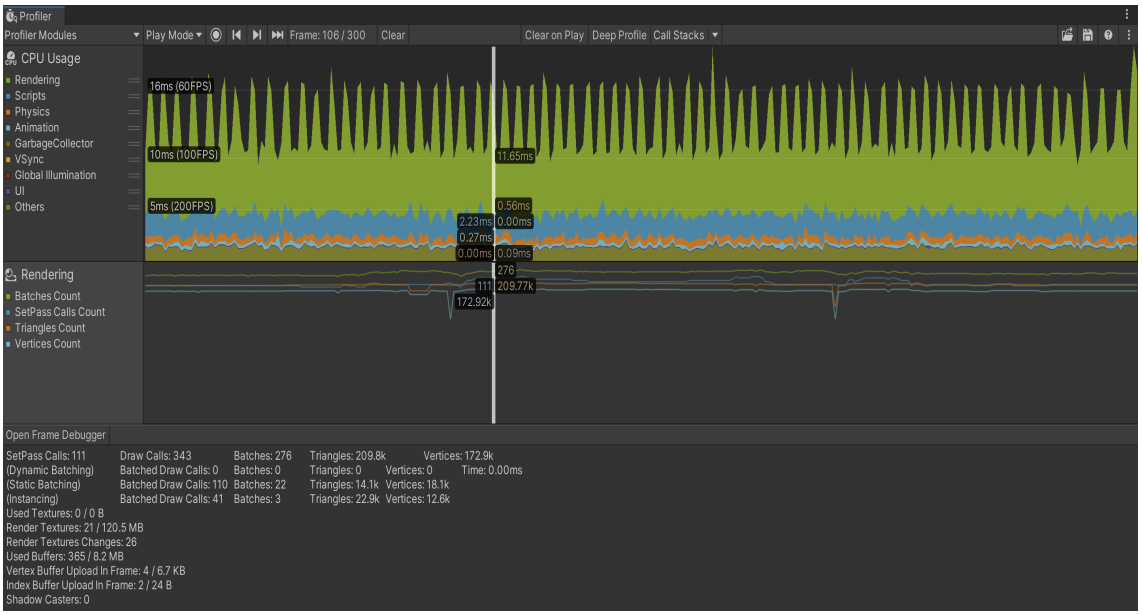


Figura 102: Atto 2, batching senza SRP Batcher, draw calls .

Analizzando la scena con il frame debugger, c'è effettivamente un riscontro del fatto che il GPU Instancing, lo static e il dynamic batching stiano funzionando (fig. 103).

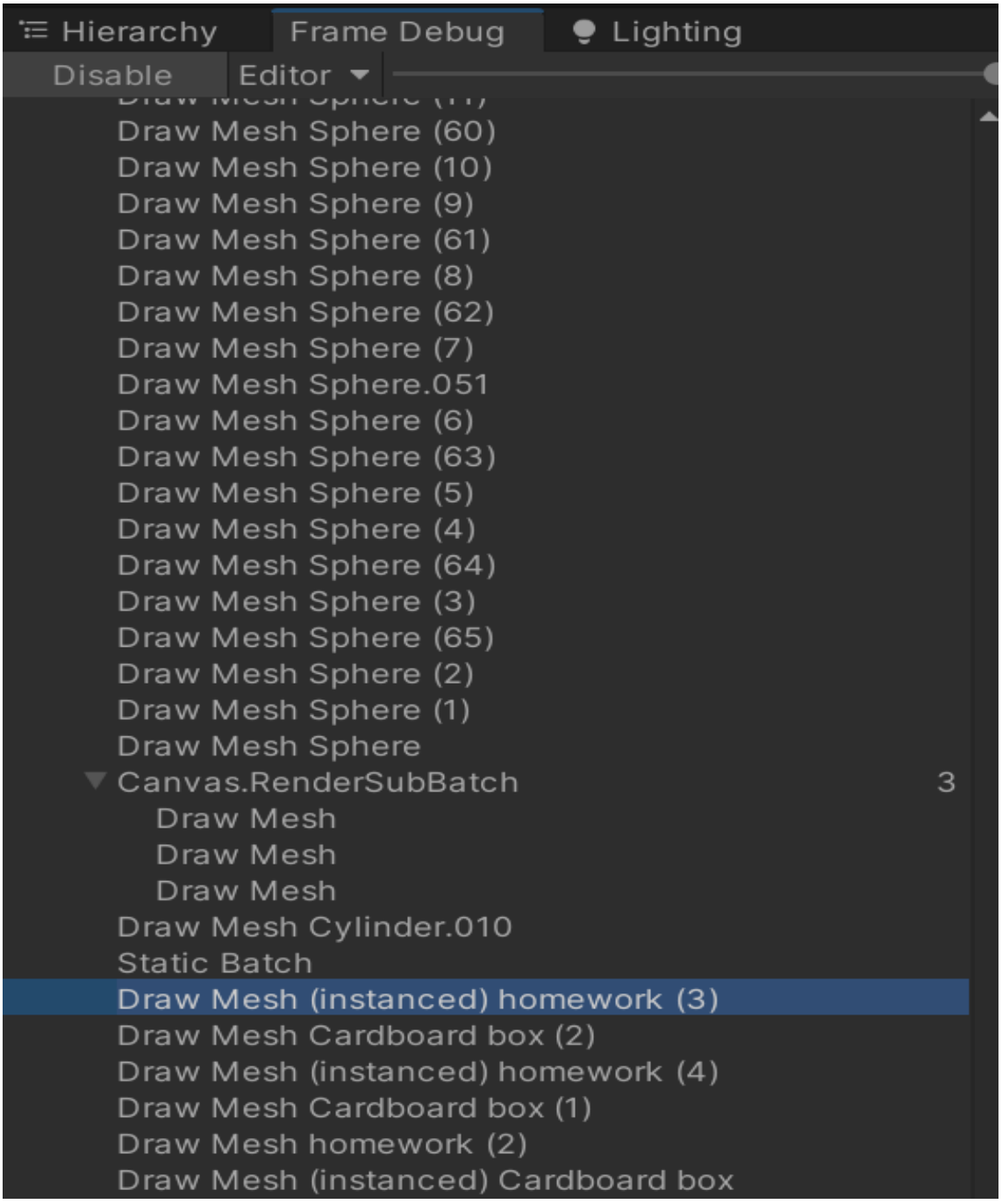


Figura 103: Atto 2, batching senza SRP Batcher, frame debugger .

8.3.3 Culling

Dopo aver effettuato il culling della scena, i tempi di processing sono rimasti pressochè identici (fig. 104 e 105).

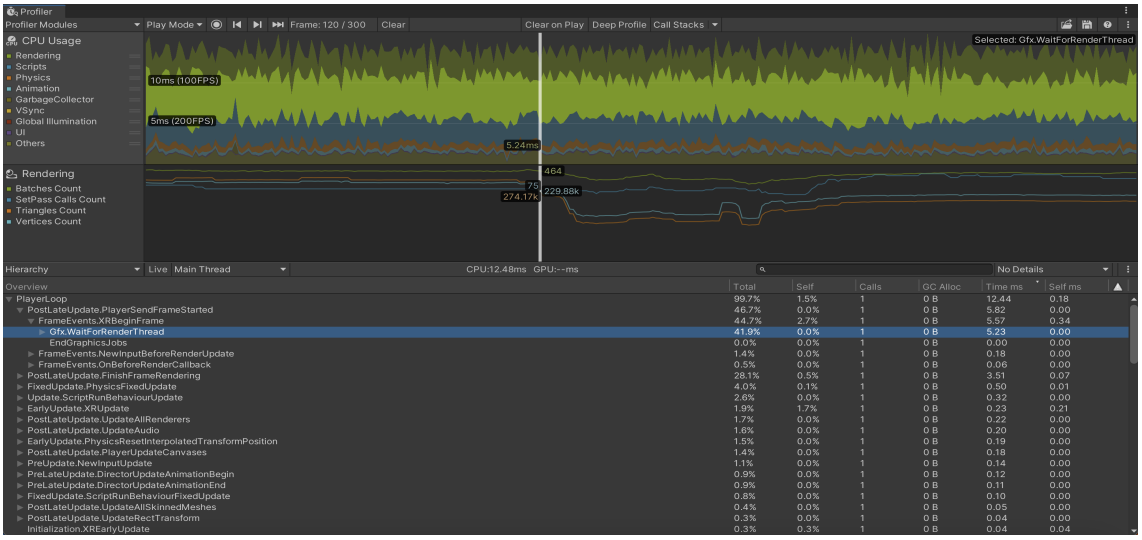


Figura 104: Atto 2, culling, Main Thread .

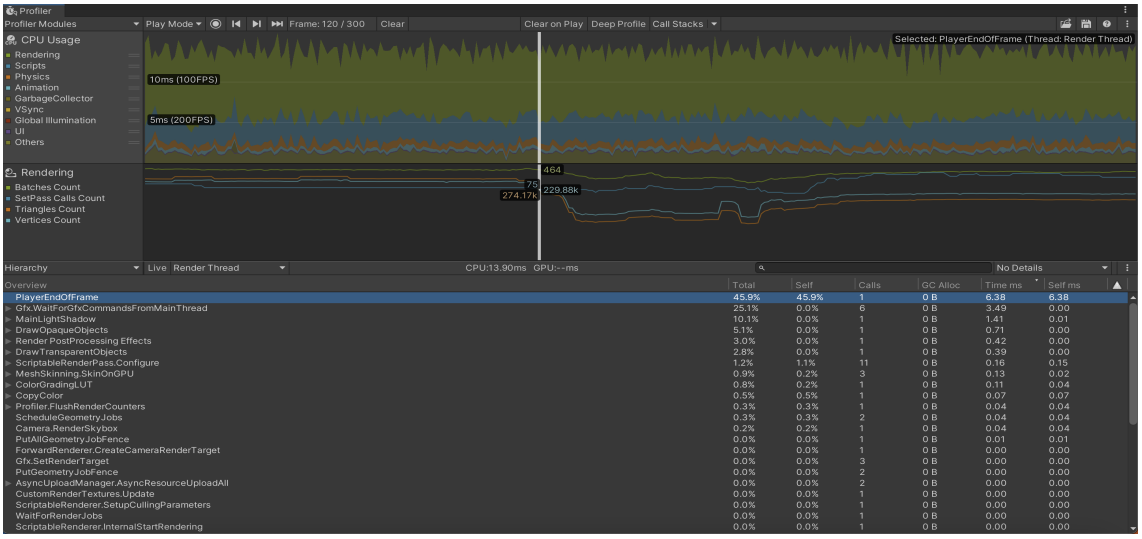


Figura 105: Atto 2, culling, Render Thread .

8.3.4 Baking

Con il baking infine, si è scesi sui 10 ms di processing (fig. 106 e 107), mentre le draw calls sono rimaste abbastanza alte (220) ma con 140 batch (fig. 108).

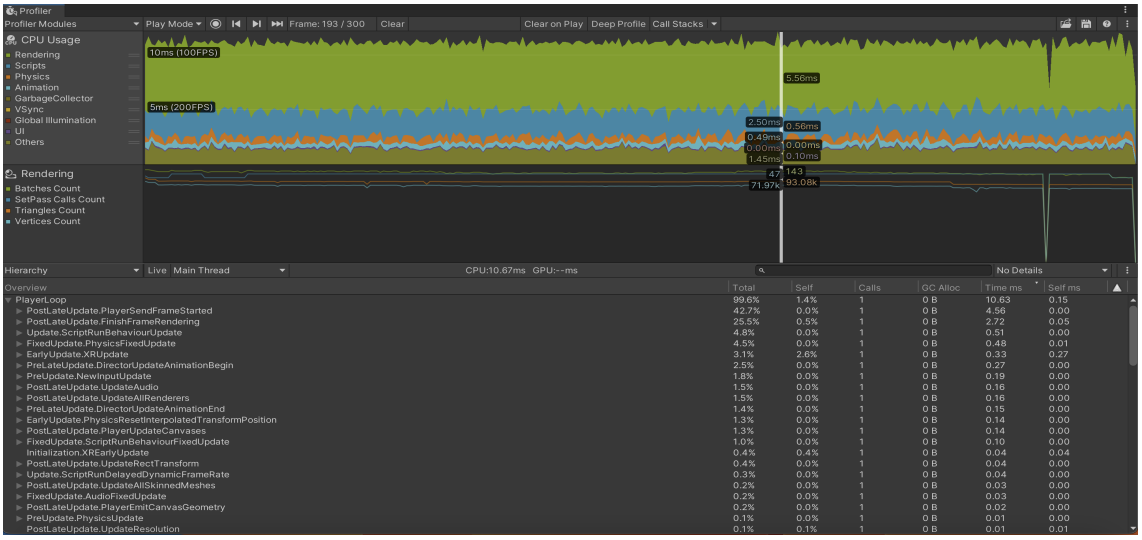


Figura 106: Atto 2, baking, Main Thread .

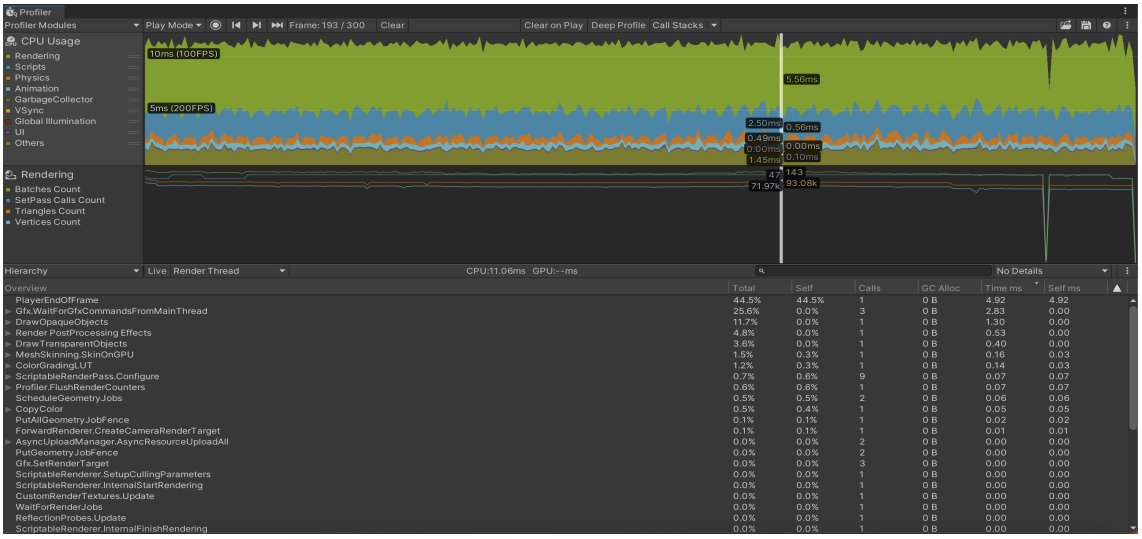


Figura 107: Atto 2, baking, Render Thread .

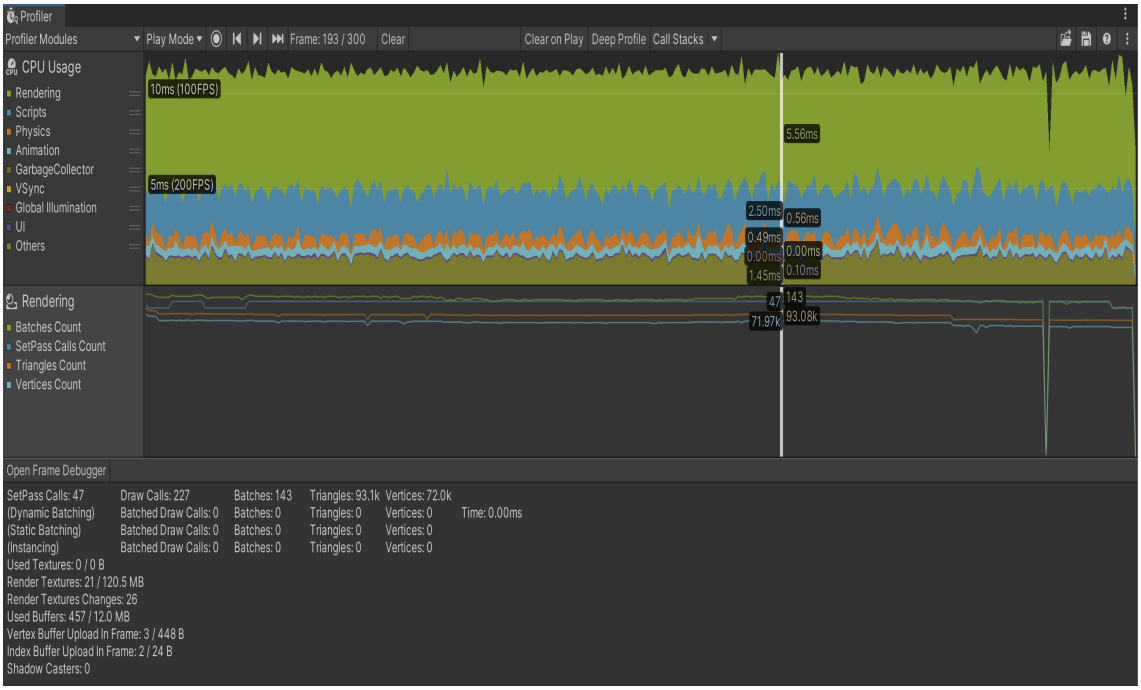


Figura 108: Atto 2, baking, draw calls .

Si ha ancora un aumento sia di draw calls che di tempi di processing quando compaiono le scatole e i compiti, il che causa un leggero frame drop (fig. 109, 110 e 111).

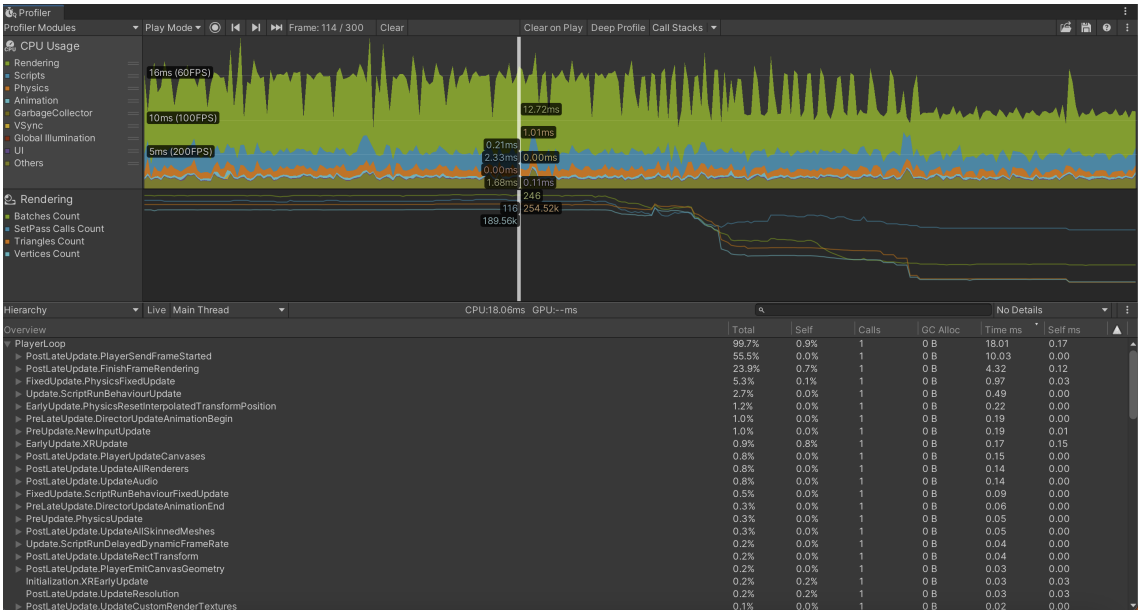


Figura 109: Atto 2, baking, main thread durante la comparsa delle scatole .

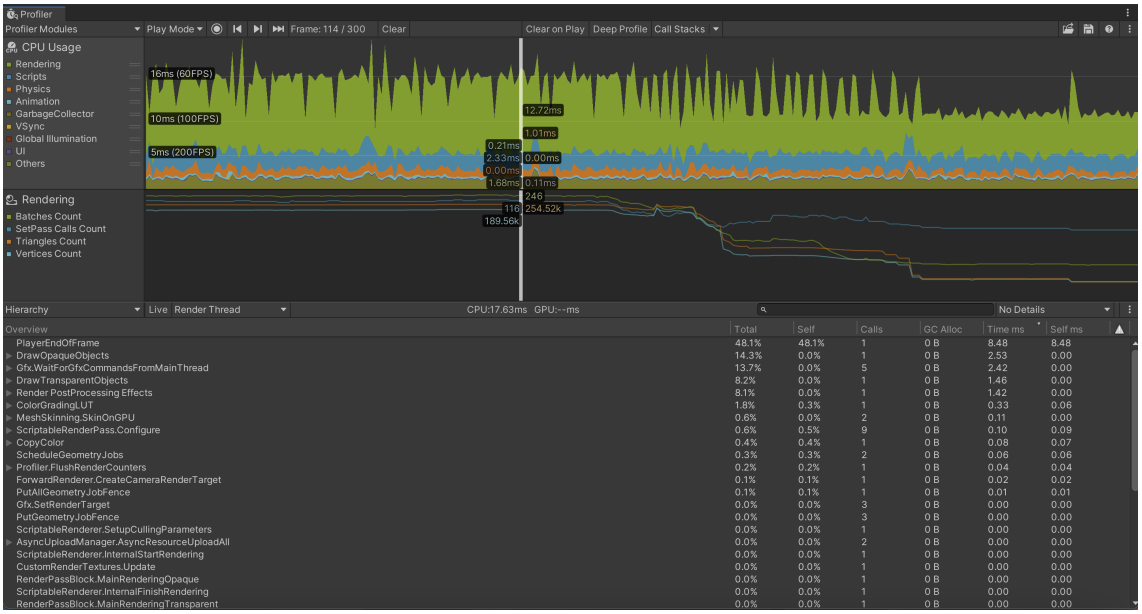


Figura 110: Atto 2, baking, render thread durante la comparsa delle scatole .

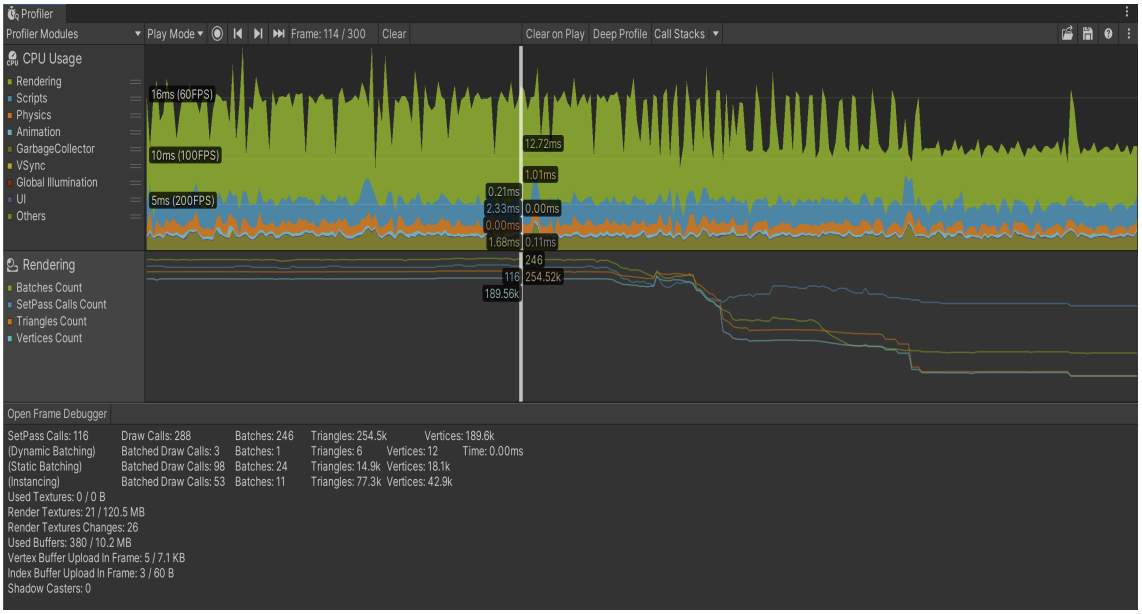


Figura 111: Atto 2, baking, draw calls durante la comparsa delle scatole .

Dopo aver abilitato l'app ad andare a 90 fps via script (prima era limitata a 72 fps), la scena è stata profilata con OVRMetricTools per verificare che venissero effettivamente raggiunti i 90 fps. Vengono mostrati qua i risultati di tre misurazioni: una a luci spente (fig. 112), a luci accese (fig. 113) e durante la comparsa di compiti e scatole (fig. 114).

Figura 112: Atto 2, baking, misurazione con OVR a luci spente .

Figura 113: Atto 2, baking, misurazione con OVR a luci accese .

Figura 114: Atto 2, baking, misurazione con OVR durante la comparsa di compiti e scatole .

La scena continua ad avere un notevole drop di fps quando compaiono scatole e compiti.

8.3.5 Combine Meshes

Applicando la tecnica della combinazione delle singole mesh in una mesh più grande per le mesh dei compiti e delle scatole, il drop che si aveva precedentemente (fig. 114), che portava il frame rate a 63 fps, viene quasi completamente risolto: adesso il frame rate si attesta sugli 82-83 fps (fig. 115).

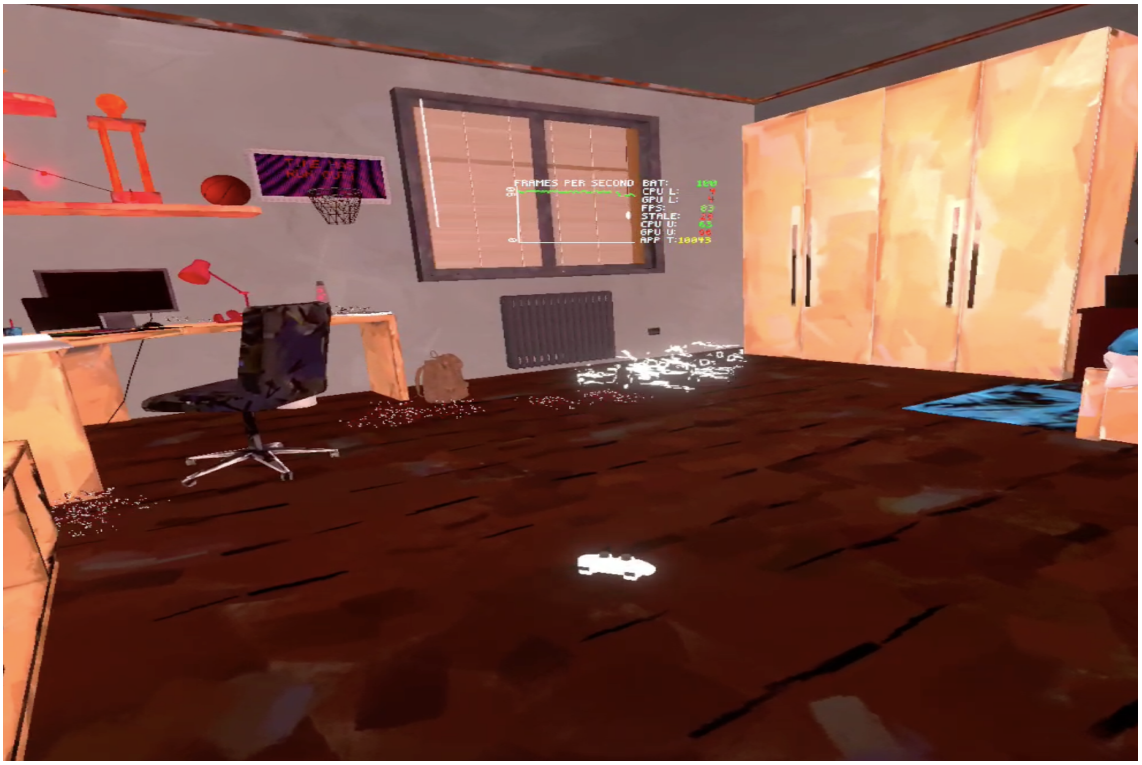


Figura 115: Atto 2, baking, misurazione con OVR durante la comparsa di compiti e scatole .

Capitolo 9

Conclusioni

In conclusione, in questo elaborato sono stati elencati tutti i metodi e i tool per profilare un'app, individuare gli eventuali colli di bottiglia e risolverli, applicando poi tutte le tecniche spiegate nei primi capitoli a un'applicazione che presentava diversi problemi di lentezza. Le tecniche di ottimizzazione utilizzate si son rivelate estremamente efficaci, portando l'atto 1 di LockedUp da 25-30 fps a 90 fps e il secondo atto da 40-45 a 90 fps, ben sopra l'obiettivo minimo di 72 fps indicato da Meta.

Bibliografia

[1] Meta. Specifiche di oculus quest 2. https://developer.oculus.com/resources/oculus-device-specs/#meta-quest-2. Accessed: 2023-01-08.

[2] Meta. Performance di riferimento per un'app per oculus quest 2. https://developer.oculus.com/documentation/unity/unity-perf/. Accessed: 2023-01-08.

[3] Unity. Xr interaction toolkit. https://docs.unity3d.com/Packages/com.unity.xr.interaction.toolkit@2.3/manual/index.html. Accessed: 2023-01-08.

[4] Unity. Unity profiler. https://docs.unity3d.com/Manual/Profiler.html. Accessed: 2023-01-08.

[5] Unity. *Ultimate Guide To Profiling Unity Games*. Unity, 2022.

[6] Unity. Frame debugger. https://docs.unity3d.com/Manual/FrameDebugger.html. Accessed: 2023-01-08.

[7] RenderDoc. Render doc. https://renderdoc.org. Accessed: 2023-01-08.

[8] Meta. Ovr metrics tool. https://developer.oculus.com/downloads/package/ovr-metrics-tool/. Accessed: 2023-01-08.

[9] Unity. Unity profiler windows. https://docs.unity3d.com/Manual/ProfilerWindow.html/. Accessed: 2023-01-08.

[10] Chris Language Jonathan Parham Aleksandra Kizevska Eric Van de Kerckhove Ajay Venkat, Sandra Grauschopf. Unity job system. https://www.kodeco.com/7880445-unity-job-system-and-burst-compiler-getting-started%5C#toc-anchor-011. Accessed: 2023-01-08.

[11] Unity. Cos'è il job system. https://blog.unity.com/engine-platform/what-is-a-job-system. Accessed: 2023-01-08.

[12] Clay Slover. Lod. https://www.artstation.com/blogs/samslover/QwNE/ue4-optimization-performance-pt2-lods. Accessed: 2023-01-08.

[13] Blender Artist. Lod. https://blenderartists.org/t/3d-environment-scans-to-lod-game-assets/626400. Accessed: 2023-01-08.

[14] Unity. Lod in unity. https://docs.unity3d.com/Manual/LevelOfDetail.html. Accessed: 2023-01-08.

[15] Unity. Lod group in unity. https://docs.unity3d.com/Manual/class-LODGroup.html. Accessed: 2023-01-08.

[16] IDV Inc. Speed tree. https://store.speedtree.com. Accessed: 2023-01-08.

[17] Wikipedia. Speed tree (wiki). https://en.wikipedia.org/wiki/SpeedTree. Accessed: 2023-01-08.

[18] Unity. Speed tree in unity. https://unity.com/products/speedtree. Accessed: 2023-01-08.

[19] ARM Developer. Draw calls, best practice. https://developer.arm.com/documentation/101897/0301/Optimizing-application-logic/Draw-call-batching-best-practices. Accessed: 2023-01-08.

[20] Unity. Optimize draw calls in unity. https://docs.unity3d.com/Manual/DrawCallBatching.html. Accessed: 2023-01-08.

[21] Unity. Static batching. https://docs.unity3d.com/Manual/static-batching.html. Accessed: 2023-01-08.

[22] Unity. Combine meshes. https://docs.unity3d.com/ScriptReference/Mesh.CombineMeshes.html. Accessed: 2023-01-08.

[23] Unity. Gpu instancing. https://docs.unity3d.com/Manual/GPUInstancing.html. Accessed: 2023-01-08.

[24] Unity. Occlusion culling. https://docs.unity3d.com/Manual/OcclusionCulling.html. Accessed: 2023-01-08.

[25] Unity. Lightbaking. https://unity.com/how-to/advanced/optimize-lighting-mobile-games#lighting-challenges-mobile. Accessed: 2023-01-08.

[26] Unity. Raycast command. `https://docs.unity3d.com/ScriptReference/RaycastCommand.html?utm_source=demand-gen&utm_medium=pdf&utm_campaign=asset-links-gmg-choose-unity-for-multiplatform&utm_content=optimize-game-performance-2020-lts-ebook`. Accessed: 2023-01-08.

[27] Unity. Ottimizzazione della fisica. `https://blog.en.uwa4d.com/2022/03/01/unity-performance-optimization--physics/`. Accessed: 2023-01-08.

[28] Unity. Collision callbacks. `https://docs.unity3d.com/ScriptReference/Physics-reuseCollisionCallbacks.html`. Accessed: 2023-01-08.