

# **Profilazione e tecniche di ottimizzazione**

Indice	pag
<b>1 - Introduzione Android Profiler</b>	<b>3</b>
1.1 – Android GPU Inspector	3
<b>2 - Funzionamento Android Profiler</b>	<b>4</b>
2.1 - Analisi CPU	4
2.1.1 - Utilizzo della CPU	6
2.1.2 - Lista dei Thread	7
2.1.3 – Recording	7
2.1.4 - Dati ritornati dal Recording	8
2.2 - Analisi RAM	10
2.3 - Analisi Network	10
2.4 - Analisi Consumo energetico	10
<b>3 - Descrizione App realizzata</b>	<b>12</b>
3.1 - Cenni sulla realizzazione	12
<b>4 - Rendering App</b>	<b>13</b>
4.1 - Oggetto View e inserimento nel layout dichiarativo	13
4.2 - Realizzazione Renderer	14
4.3 - Utilizzo della OpenGL	14
4.3.1 – Disegnare un cubo con la GLES20	14
<b>5 - Prestazioni prima stesura</b>	<b>15</b>
5.1 - Importanza ADT	15
5.2 - Primi accorgimenti nel Renderer	15
5.3 - Prima analisi prestazionale	16
5.3.1 - Profilazione Emulatore	16
5.3.2 - Profilazione dispositivo fisico	21
5.4 - Rilevazione di Bottleneck	26
<b>6 - Tecniche di ottimizzazione</b>	<b>27</b>
<b>7 - Considerazioni finali</b>	<b>30</b>
7.1 - Risultati ottimizzazione	30

# 1 Introduzione Android Profiler

Il Profiler di Android Studio è un insieme di strumenti che forniscono dati in tempo reale per aiutare a capire in che modo l' app utilizza:

- CPU
- RAM
- Rete
- Batteria

Il profiler viene usato per determinare l'uso delle varie parti dell'hardware e trovare poi le cause di eventuali bottleneck.

Viene utilizzato non solo per usufruire l'hardware al massimo delle capacità, come nel caso dei giochi o altre app che necessitano di prestazioni elevate, ma anche per app disegnate che usino meno risorse possibili, in particolar modo l'utilizzo della batteria.

## 1.1 Android GPU Inspector

Difetto del Profiler di Android Studio è che non esegue nessuna profilazione per la GPU, per eseguire quest'ultima si dovrebbe usare un programma esterno: [Android GPU Inspector \(AGI\)](#)


AGI permette di acquisire una traccia da un singolo frame dell'applicazione e quindi eseguire un'analisi approfondita dell'utilizzo della GPU del gioco.

Ciò include una copertura e un'analisi più approfondite delle API Vulkan e OpenGL ES

**Non approfondisco questa applicazione poiché NON ho potuto utilizzarla causa delle sue limitazioni di utilizzo: [Link](#)**

## 2 Funzionamento

Android Profiler è compatibile con Android 5.0 (livello API 21) e versioni successive.

Quando si vuole profilare l'app, bisogna eseguirla tramite Android Studio utilizzando il tool bar ► 

All'avvio dell'app, il profiler incomincerà ad acquisire dati finché non si termina la sessione del profiler o si termina l'app.

Il profiler si presenta nel seguente modo:

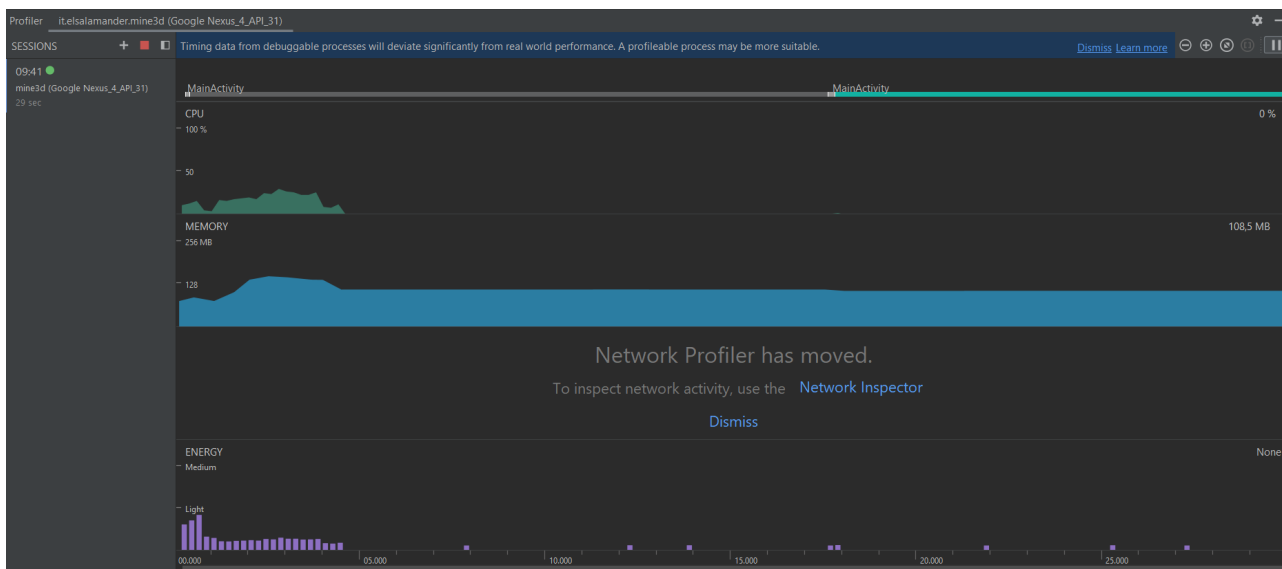


Figura 1: Vista principale del Profiler

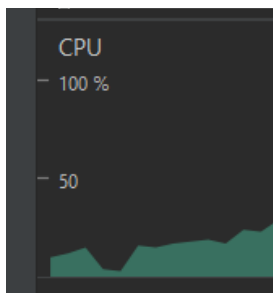


Figura 2: Utilizzo CPU

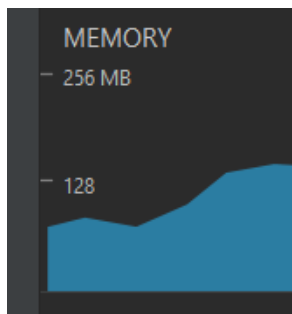


Figura 3: Utilizzo RAM

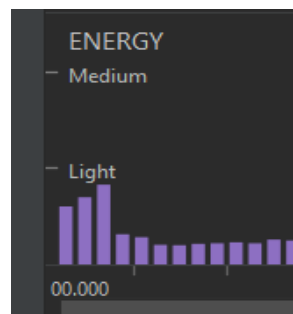


Figura 4: Utilizzo Batteria

### 2.1 Analisi CPU

È possibile utilizzare il profiler della CPU per controllare l'utilizzo di essa nell'app e l'attività del thread in tempo reale durante l'interazione con l'app.

Inoltre è possibile controllare i dettagli nelle tracce dei metodi, delle funzioni e del sistema registrate.

Le informazioni dettagliate che il Profiler CPU registra e mostra, sono determinate dalla configurazione di registrazione scelta:

- **Traccia del sistema:** acquisisce dettagli finemente che consentono di ispezionare il modo in cui l'app interagisce con le risorse di sistema.
- **Tracce di metodi e funzioni:** per ogni thread nel processo dell'app, ritorna quali metodi (Java/Kotlin) o funzioni (C/C++) vengono eseguite in un periodo di tempo e le risorse della CPU che ogni metodo o funzione consuma durante la sua esecuzione.  
È inoltre possibile utilizzare le tracce di metodi e funzioni per identificare il **chiamante** e **chiamato**.

Un **chiamante** è un metodo o una funzione che invoca un altro metodo o funzione e un **chiamato** è uno che viene invocato da un altro metodo o funzione.

Si può usare queste informazioni per determinare quali metodi o funzioni sono responsabili dell'invocazione, di particolari attività che richiedono molte risorse troppo spesso e, ottimizzare il codice dell'app per evitare lavori non necessari.

Quando si registrano le tracce del metodo, è possibile scegliere la registrazione **campionata** o **strumentata**. Quando si registrano tracce di funzione, è possibile utilizzare solo la registrazione campionata.

Per i dettagli sull'utilizzo e la scelta di ciascuna di queste opzioni di traccia, vedere: [Scegliere una configurazione di registrazione](#)

Come appare il profiler quando si ispeziona la CPU:

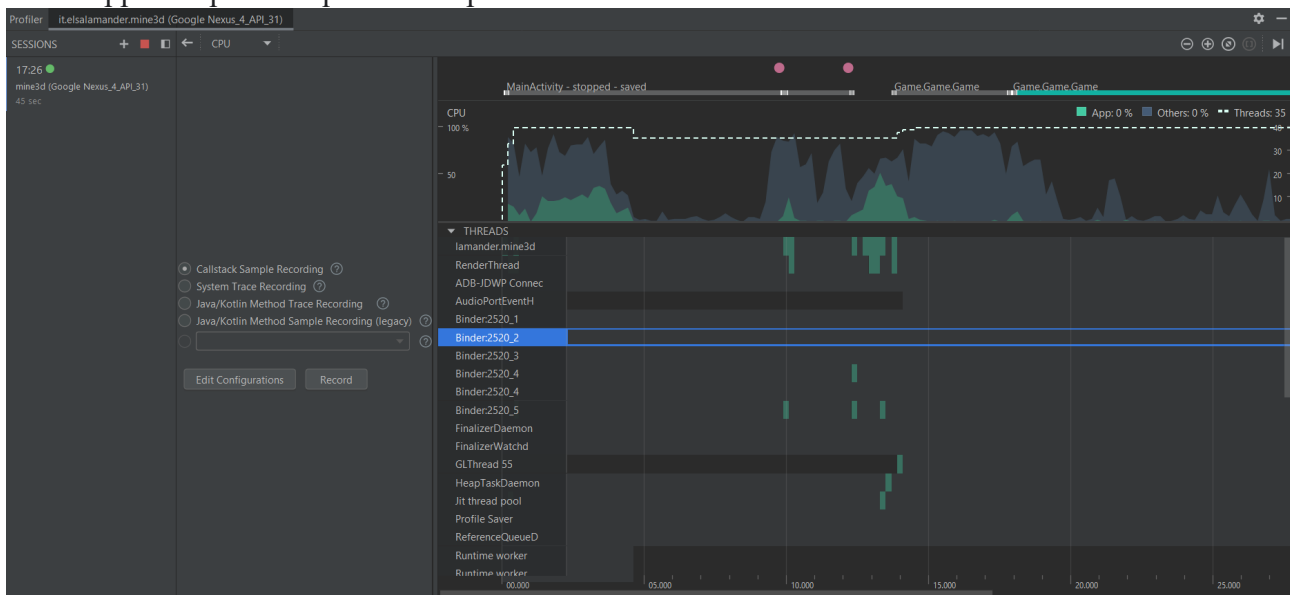


Figura 5: Utilizzo più dettagliato della CPU

Da cosa è composta questa vista:

- Utilizzo della CPU percentuale
- Lista dei processi che vengono usati e quando essi “funzionano”
- Impostazioni per la registrazione

## 2.1.1 Utilizzo della CPU

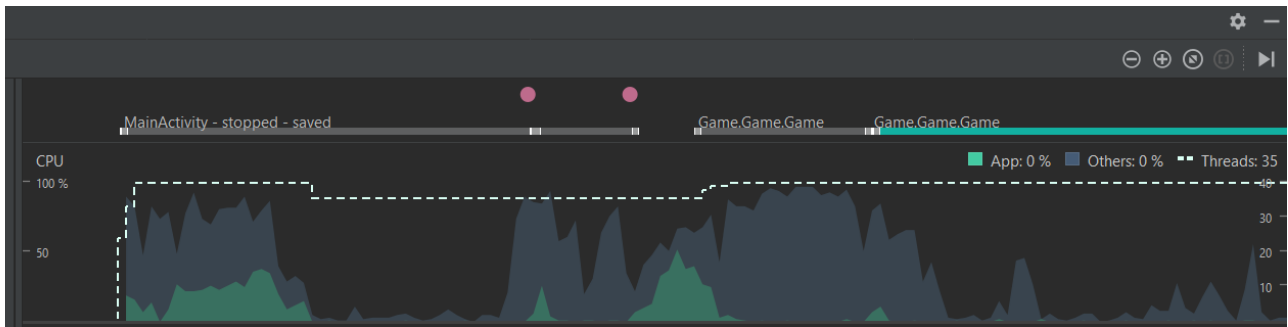


Figura 6: Utilizzo percentuale della CPU più dettagliato

Questa è la prima parte che verrà discussa in quanto già questo grafico mostra informazioni utili non banali, tra cui:

- Utilizzo percentuale dovuto dall'app in **VERDE**.
- Utilizzo percentuale non dovuto dall'app in **GRIGIO**.
- Numero dei Thread presenti durante l'evolversi degli eventi.

Sopra i grafici ci sono altre 2 Righe:

- Una in cui ci sono 2 pallini rossi ● che mostra quando si tocca lo schermo.
- Una in cui si mostra il ciclo di vita delle Activity.

Non è banale perché bisogna considerare un fattore hardware per leggere e interpretare al meglio questo grafico, ovvero il numero di Core presenti nella CPU.

Si potrebbe pensare che l'utilizzo del 25% della CPU sia basso rispetto al totale, invece **NO**, poiché **se la funzione non è multithreading, essa non sfrutterà mai il 100% della CPU** ma solo il 25%, (25% nel mio caso perché nell'emulatore ho messo 4 core), pensare che l'utilizzo della CPU è basso e quindi che non crea bottleneck è un grave errore di lettura e interpretazione.

## 2.1.2 Lista dei Thread

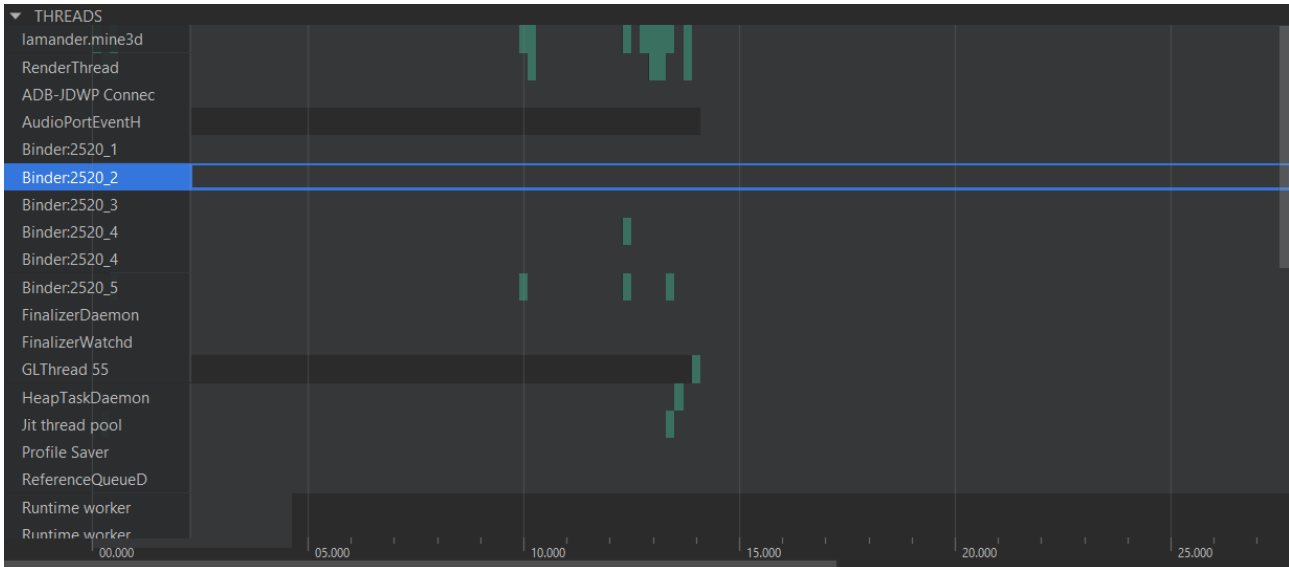


Figura 7: Lista dei Thread presenti durante l'esecuzione dell'app

Sotto al grafico che mostra l'utilizzo percentuale della CPU c'è la lista dei Thread presenti.

Il grafico, sopra citato (Figura 6) è il raggruppamento delle informazioni che vengono mostrate da questo grafico (Figura 7), che contiene per ogni Thread presente:

- “Nessun colore (=lo sfondo)” quando esso non esiste o è stato “ucciso”
- In **GRIGIO** quando il Thread è in sleeping
- In **VERDE** quando viene utilizzato

Si possono già fare delle considerazioni ovvero, che ci sono un numero consistente di Thread che vengono usati dall'app, non da parte mia, ma da parte del FrameWork di Android, più precisamente è la OpenGL che fa un certo uso di Thread diversi.

## 2.1.3 Recordign

Per avere più informazioni riguardo l'utilizzo della CPU bisogna fare un “Recording”.

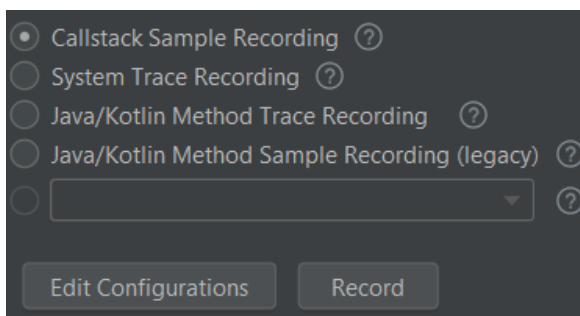
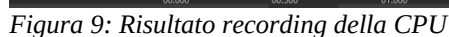


Figura 8: Impostazioni base Record

Ci sono varie impostazioni per fare un recording, al momento del bisogno, ovvero quando voglio avere una analisi più approfondita, si inizia a registrare premendo “**Record**” e poi, fermare con il pulsante “**Stop**” che sostituirà “**Record**” quando viene premuto, poiché esso, genera un file con una quantità spropositata di dati.

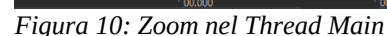
Tali dati possono poi essere eventualmente salvati e analizzati quando si vuole.

Come osservare i dati che il recording fornisce



- Una a sinistra dove è presente un grafico
- Una a destra dove sono presenti delle funzioni con alcuni parametri associati

(Ci sono talmente così tante funzioni che non mi stanno nello schermo)



Ogni riquadro rappresenta una funzione, la loro larghezza orizzontale rappresenta il tempo di esecuzione, la funzione chiamante è quella in alto.



Il riquadro a destra, è un altro modo di raffigurare le stesse informazioni, però con alcuni vantaggi, ovvero, mette sotto la stessa funzione, la somma di tutti i tempi che quella funzione ci mette per essere realizzata, nel caso che la funzione in questione viene chiamata più volte.

Name	Total (µs)	%	Self (µs)	%	Children (µs)
> m main() ()	2.760.701	100,00	66	0,00	2.760.635
> m Binder:2597_10 ()	2.760.701	100,00	2.748.875	99,57	11.826
> m Studio:InputCon0 ()	2.760.701	100,00	154	0,01	2.760.547
> m Binder:2597_40 ()	2.609.973	94,54	2.603.371	94,30	6.602
> m RenderThread0 ()	2.065.677	74,82	2.065.136	74,80	541
> m Thread-70 ()	304.772	11,04	2	0,00	304.770
> m ReferenceQueueDae	6.569	0,24	8	0,00	6.561
> m SetSubtitleAnchorTh	1.631	0,06	16	0,00	1.615
> m FinalizerDaemon0 ()	1.549	0,06	206	0,01	1.343
> m AudioPortEventHand	434	0,02	2	0,00	432
> m hwuiTask1() ()	419	0,02	2	0,00	417
> m hwuiTask0() ()	391	0,01	2	0,00	389
> m FinalizerWatchdogD	261	0,01	36	0,00	225
> m HeapTaskDaemon()	200	0,01	2	0,00	198

Figura 11: Tempi di esecuzione

Name	Total (µs)	%	Self (µs)	%	Children (µs)	%
✓ m main() ()	2.760.701	100,00	66	0,00	2.760.635	100,00
m loopOnce() (android.os.Looper)	2.760.635	100,00	590	0,02	2.760.045	99,98
m dispatchMessage() (android.os.Handler)	2.619.627	94,89	114	0,00	2.619.513	94,89
m handleMessage() (android.app.ActivityThread\$H)	2.178.722	78,92	79	0,00	2.178.643	78,92
m execute() (android.app.servertransaction.TransactionExecutor)	2.178.355	78,91	80	0,00	2.178.275	78,90
m executeCallbacks() (android.app.servertransaction.TransactionExecutor)	1.499.749	54,32	71	0,00	1.499.678	54,32
m execute() (android.app.servertransaction.LaunchActivityItem)	1.498.462	54,28	14	0,00	1.498.448	54,28
m handleLaunchActivity() (android.app.ActivityThread)	1.498.409	54,28	28	0,00	1.498.381	54,28
m performLaunchActivity() (android.app.ActivityThread)	1.491.803	54,04	57	0,00	1.491.746	54,04
m reportSizeConfigurations() (android.app.ActivityThread)	6.036	0,22	21	0,00	6.015	0,22
m handleConfigurationChanged() (android.app.ConfigurationControl	378	0,01	36	0,00	342	0,01
m <init>() (android.content.res.Configuration)	96	0,00	5	0,00	91	0,00
m preload() (android.graphics.HardwareRenderer)	14	0,00	14	0,00	0	0,00
m unscheduleGcIdler() (android.app.ActivityThread)	12	0,00	4	0,00	8	0,00
m initialize() (android.view.WindowManagerGlobal)	12	0,00	7	0,00	5	0,00
m setOldState() (android.app.servertransaction.PendingTransactionA	7	0,00	7	0,00	0	0,00
m setProfiler() (android.app.ActivityThread\$Profiler)	6	0,00	6	0,00	0	0,00
m hintActivityLaunch() (android.os.GraphicsEnvironment)	6	0,00	6	0,00	0	0,00
m getConfiguration() (android.app.ConfigurationController)	4	0,00	4	0,00	0	0,00
m setRestoreInstanceState() (android.app.servertransaction.PendingT	3	0,00	3	0,00	0	0,00
m startProfiling() (android.app.ActivityThread\$Profiler)	2	0,00	2	0,00	0	0,00
m setCallOnPostCreate() (android.app.servertransaction.PendingTran	2	0,00	2	0,00	0	0,00
m getLaunchingActivity() (android.app.ActivityThread)	23	0,00	6	0,00	17	0,00
m traceBegin() (android.os.Trace)	8	0,00	1	0,00	7	0,00
m traceEnd() (android.os.Trace)	8	0,00	3	0,00	5	0,00

Figura 12: Tempi di esecuzione - espansione

Come mostrato nel Thread “main” vengono chiamate varie funzioni in cascata, importante da sottolineare è che l’ordine con cui vengono mostrate, non centra nulla con il quando vengono chiamate, ovvero l’ordine di esecuzione, ma **vengono ordinate tenendo conto del loro tempo di esecuzione, mettendo in cima la funzione più lunga**, questo permette di trovare le funzioni più “pesanti” in termini di tempo di esecuzione.

Farò molto uso di questo, per la rilevazione della bottleneck.

## 2.2 Analisi RAM

E' possibile profilare l'uso della RAM utilizzata dall'app, per sapere quanta di essa viene occupata nell'evolversi degli eventi, differenziando gli usi come mostrato dall'immagine.

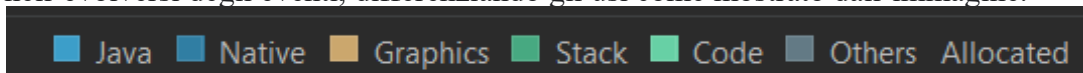


Figura 13: Diversificazione della RAM occupata

Come mostrato nella figura, che è la leggenda del grafico, la RAM utilizzata viene differenziata nei vari utilizzi tra cui:

1. Java: Memoria che viene allocata tramite codice Java o Kotlin.
2. Native: Memoria che viene allocata tramite codice C o C++.
3. Graphics: Memoria utilizzata dai Buffer grafici, texture e altri oggetti della OpenGL.
4. Stack: Memoria utilizzata da Java/Kotlin + C/C++ per via dei vari Thread presenti.
5. Code: Memoria utilizzata per varie risorse, che possono essere librerie.so o bytecode.
6. Others: Memoria che non si riesce a classificare.
7. Allocated: Il numero di oggetti allocati in Java/Kotlin non conta quelli allocati in C/C++.

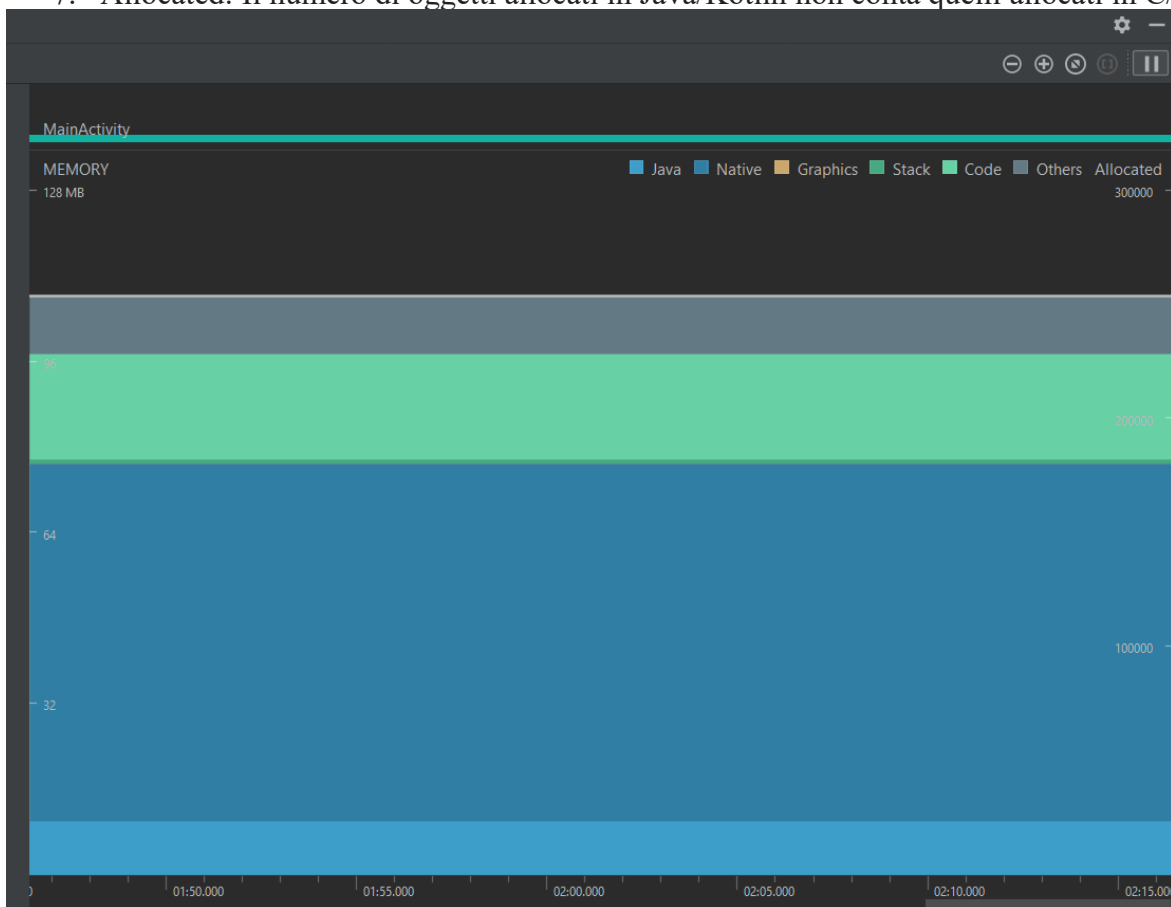


Figura 14: Profilo utilizzo della RAM

Questo è un profilo della RAM occupata da una app, come per la CPU si può eseguire un Recording per ispezionare più nel dettaglio l'utilizzo di essa, per scoprire come è frammentata la RAM e quindi come è composto l'Heap in particolare modo.

**Non approfondisco la profilazione della RAM** poiché non faccio un enorme uso e durante l'esecuzione, non vado a creare e distruggere oggetti con una frequenza elevata generando un Heap che poi sarebbe stato il caso di controllare.

Inoltre, sempre nel mio caso che la RAM occupata è poca in confronto ai quantitativi che hanno mediamente i nuovi telefoni, non temo in particolare modo terminazioni dell'app forzate dal sistema per recuperare risorse.

Questa è una profilazione necessaria per applicazioni che fanno grande uso della RAM poiché, bisogna considerare il caso in cui l'app va messa in background, creando un possibile calo di prestazioni dovuto dalla RAM eccessivamente occupata.

Un altro motivo per cui non approfondisco la profilazione della RAM, è che non intacca in modo consistente le performance della mia applicazione.

## 2.3 Analisi NetWork

E' possibile visualizzare in tempo reale i dati inviati e ricevuti dall'applicazione.

Poiché la mia app non si connette a internet e quindi non ha nessun tipo di scambio dati con il Network, **non approfondisco questo tipo di profilazione**.

## 2.4 Analisi Consumo energetico

E' possibile visualizzare, nel corso degli eventi, il livello di consumo energetico dovuto dall'applicazione, un aspetto molto richiesto per i giochi e per le applicazioni in background perché mostra i livelli di consumo energetici.



Figura 15: Consumo energetico

Il consumo dipende principalmente dall'uso della CPU e dalla "scheda di Rete".

Come per la CPU, RAM e per il NetWork si può fare un Recording per avere più informazioni riguardo il consumo energetico, in modo tale, da identificare il pezzo di codice/funzione in un certo l'asso di tempo T che assorbe l'energia mostrata.

Questo studio è importante per le app in background le quali non dovrebbe intaccare la durata della batteria e quindi avere il consumo energetico più basso possibile e da tutte le applicazioni che utilizzano al massimo l'hardware e software come i giochi di una certa fascia.

Come per la RAM **non approfondisco** questa profilazione.

### 3 Descrizione App realizzata

L'app realizzata, è un gioco ispirato al gioco già esistente scaricabile dal PlayStore "MasterMine". Il gioco è la realizzazione del classico gioco "Campo minato" ma versione 3D, ovvero, non è una griglia 2D ma ben si 3D, le "celle" che diventano "Cubi" sono sulla superficie di un "Grande Cubo".

L'app fornisce varie configurazioni tra cui:

- Tema in game
- Comando per svelare
- Comando per mettere la bandiera
- Sensibilità per la rotazione dei 3 assi
- Tempo di Hold per il comando "Hold"
- Mostra il Timer in game
- Mostra le "bombe" rimanenti (in realtà conta all'indietro le bandiere piazzate)
- Vibrazione quando si perde
- Livello musica
- Livello effetti speciali

Sono presenti alcuni preset per i controlli.

Il gioco poi ha altre impostazioni:

- Grandezza lato della griglia
- Percentuale bombe presenti
- Quando si vince incrementare o no il lato della griglia
- Quanto incrementare la griglia

Anche qua sono presenti dei preset per la Difficoltà > Percentuale bombe presenti

Quando si vuole avviare un game, si può fare uso di 6 livelli standard + la possibilità di customizzare il game tramite le voci sopra citate, in aggiunta, c'è la possibilità di recuperare il livello precedente se non era stato concluso.

#### 3.1 Cenni sulla realizzazione

Descrizione di come è stata realizzata l'app senza entrare nel dettaglio di come è stata scritta tutta l'applicazione.

Solo note che verranno riprese più avanti.

E' costituita da **2 Activity**, **ogni activity poi è composta da fragment**, la struttura è:

1 - Activity 1:

- 1.1 - Fragment Menu principale
- 1.2 - Fragment Settings principali
- 1.3 - Fragment Settings controlli
- 1.4 - GameStandard

2 - Activity 2:

- 2.1 - GameCustom Settings
- 2.2 - GameRender
- 2.3 - Pause
- 2.4 - EndGame

## 4 Rendering App

Descrizione di come viene eseguito il rendering nel fragment dedicato alla schermata di gioco.

### 4.1 Oggetto View e inserimento nel layout dichiarativo

Per realizzare una schermata dove poter disegnare quello che voglio, seguendo il design di Android, è stato creato un oggetto che estende la classe **GLSurfaceView**, un altro aspetto non da poco è la firma del costruttore:

```
class MyGLSurfaceView(cont : Context, attrs : AttributeSet) : GLSurfaceView(cont, attrs)
```

Figura 16: Firma costruttore della classe che estende GLSurfaceView

Questa firma deve esserci così si può utilizzare tranquillamente l'oggetto nel layout.xml che descrive in modo dichiarativo il fragment, come mostrato.

```
<it.elsalamander.mine3d.Game.Graphic.Engine.MyGLSurfaceView
    android:id="@+id/glSurfaceViewID"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent"
/>
```

Figura 17: Uso della classe MyGLSurfaceView nel file .xml

Questo mi permette di usare i vari attributi quali “android” e “app”.

Un altro motivo per cui ho realizzato il Layout in modo dichiarativo è per inserire in modo comodo altri oggetti grafici quali:

- 2 TextView
- ImageButton

(Vedi il file “fragment\_game.xml”).

Questo serve per definire l'area dello schermo nella quale eseguo il mio rendering.

Il render deve essere settato nella GLSurfaceView la quale gestisce le chiamate di creazione della superficie, cambiamento della superficie e di disegno di essa, queste ultime 3 funzioni sono funzioni da realizzare nella classe che realizza il Rendering.

## 4.2 Realizzazione Renderer

L'oggetto che realizza il render del gioco deve implementare l'interfaccia:

**GLSurfaceView.Renderer** tale interfaccia impone la realizzazione di 3 funzioni:

```
override fun onSurfaceCreated(gl: GL10?, config: EGLConfig?) {...}

override fun onSurfaceChanged(glUnused: GL10, width: Int, height: Int) {...}

override fun onDrawFrame(gl: GL10?) {...}
```

Figura 18: Funzioni da realizzare

Le funzioni sono:

1. **onSurfaceCreated** : Viene chiamata solo al momento della creazione del layout è server per fare operazioni preliminari, tra cui:
  - Impostare lo sfondo
  - Impostare alcuni attributi della OpenGL
  - Creazione del programma per la OpenGL
  - Caricamento delle Texture che verranno usateTutte operazioni le quali necessitano solo una esecuzione, vengono effettuate qua.
2. **onSurfaceChanged**: Viene chiamata quando la grandezza dello schermo cambia, viene chiamata anche alla creazione della SurfaceView.
3. **onDrawFrame**: Viene chiamata ogni qual volta viene chiesto di eseguire un rendering.

## 4.3 Utilizzo della OpenGL

Come supporto per il rendering è stata usata la GLES20 di android.

(Per eseguire un render semplice e performante potrebbe essere buono, ma per grafiche complesse, dopo aver realizzato l'applicazione lo sconsiglio, meglio usare un supporto più ricco come Unity o UnrealEngine).

### 4.3.1 Disegnare un cubo con la GLES20

Non si disegna in modo diretto un cubo, ma ben sì, un insieme di triangoli 2D nello spazio che formeranno poi un cubo.

Ci si aiuta con matrici non piccole per scrivere una volta per tutte la posizione relativa dei vertici dei 12 triangoli che compongono un cubo.

Oltre a queste coordinate, bisogna avere un set di coordinate che compongono una matrice lunga 48 per le texture che verranno applicate ai cubi.

Servono altri Array per contenere gli ID delle texture caricate e altri valori della OpenGL allocati quando viene creata la SurfaceView.

Durante il rendering per ogni cubo, viene calcolata la sua posizione nello spazio 3D e poi, tramite tutte le matrici/vettori e altri valori precedentemente creati, si completa il rendering utilizzando esclusivamente funzioni della GLES20.

Per separare la parte di codice “costante” da quella un po più dinamica, ho **separato il rendering in 2, una classe per calcolare la posizione assoluta del cubo nello spazio**, e una **classe di tipo “object” dove viene finalizzato il rendering** utilizzando come citato, solo le funzioni della OpenGL.

## 5 Prestazioni prima stesura

Dopo che il codice è stato concluso per realizzare grosso modo l'applicazione, è stata realizzata una profilazione per cercare una eventuale bottleneck.

Prima di mostrare i risultati però, bisognerebbe discutere delle scelte fatte a inizio progetto, per realizzare fin da subito, una applicazione minimamente ottimizzata.

Gli accorgimenti principali sono dati da:

1. ADT utilizzato per assegnare a un punto 3D un oggetto X
2. Accorgimento nel rendering.

### 5.1 Importanza ADT

L'ADT ha un impatto particolare, perché tiene in modo organizzato mediando prestazioni e RAM occupata finale, un punto nello spazio 3D, in realtà può essere utilizzato per organizzare un qualsiasi punto su un spazio N-Dimensionale, nel mio caso, mi bastava avere uno spazio 3D.

Come è stato fatto?

La mia è una personale realizzazione ispirata dall'albero **quadramentale** e il **BVH**

(Il BVH è un algoritmo molto usato nei “veri” motori grafici, per il ray tracing in particolar modo per le ombre, per credere vedi qua: <https://worldoftanks.eu/en/news/general-news/ray-tracing/> , nel minuto 3:20 mostra come viene usato il BVH e il suo vantaggio nell'usarlo)

Il mio ADT, organizza i cubi in un modo simile, non uguale al BVH.

Questo mi permette di avere complessità  $O(\log(n))$  quando faccio in get di un cubo date le coordinate (x,y,z), essendo un albero è anche facile la visita dei nodi che è  $O(n)$  con “n” il numero di cubi presenti nella griglia di gioco.

Trovo che sia un ottimo compromesso per avere in fine prestazioni a livello computazionale che a livello di RAM occupata.

### 5.2 Primi accorgimenti nel Renderer

Come mostrato nell'immagine (Figura 18) si può notare che tutti i cubi hanno in comune alcuni aspetti grafici per costruire la matrice di proiezione i quali sono:

1. Matrice di rotazione:
  - Rotazione asse X
  - Rotazione asse Y
  - Rotazione asse Z
2. Posizione della camera
3. Matrice di proiezione

Per ottenere la matrice finale che combina queste informazioni grafiche in comune, bisogna compiere numerosi calcoli sulle matrici i quali sono:

- Rotazioni
- Moltiplicazioni
- Copia di matrici

Queste funzioni, che all'interno per essere eseguite usano numerosi cicli “for” lunghi quanto la matrice (le matrici in questione sono lunghe 16), aver **raggruppato questi calcoli in un singolo step anziché calcolarlo per ogni singolo cubo porta ad avere un rendering più veloce.**

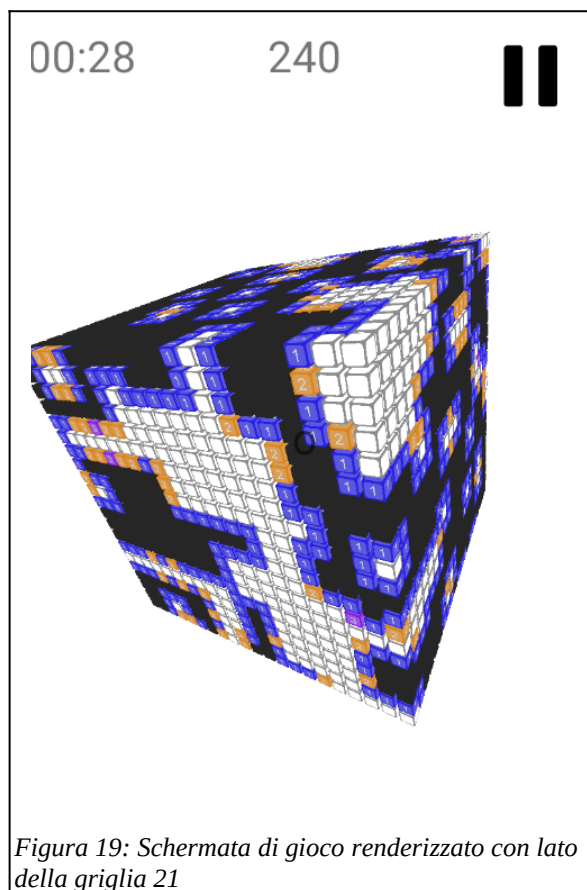


Figura 19: Schermata di gioco renderizzata con lato della griglia 21

## 5.3 Prima analisi prestazionale

Studio delle performance ottenute dalla prima stesura dell'applicazione, considerando già alcuni accorgimenti fatti quando si è creato il progetto, citati sopra.

L'analisi si concentra sull'uso della CPU durante il rendering del gioco usando:

- Emulatore
- Telefono fisico

### 5.3.1 - Profilazione Emulatore

La profilazione è stata eseguita con le seguenti impostazioni:

- Impostazioni Virtual Device:
  - Nome: Nexus 4
  - Grandezza schermo: 4,7"
  - Risoluzione: 768x1280, densità: xhdpi
  - Versione Android: S (Android 12.0 x86\_64)
  - Processore: 4 Core (Il pc di emulazione ha un i5-12600 che ha 6/12 @ 3.30 Ghz)
  - Ram: 4Gb (Il pc di emulazione ha 16 Gb DDR4 3200Mhz)
  - Scheda grafica per il render: NVIDIA GeForce RTX 3060 Laptop
- Impostazioni gioco:
  - Lato del cubo renderizzato è 7, ciò comporta un numero totale di cubi di 218 cubi

Tutti valori da tenere in considerazione poiché influenzano le performace finali ottenute.

Utilizzo della CPU quando si avvia il gioco, da quando si preme il pulsante di avvio game fino al suo primo render, di fatti nella riga dedicata al ciclo di vita delle activity si nota la conclusione dell'activity del "menu" e inizio di quella del "game" dove è presente il render.

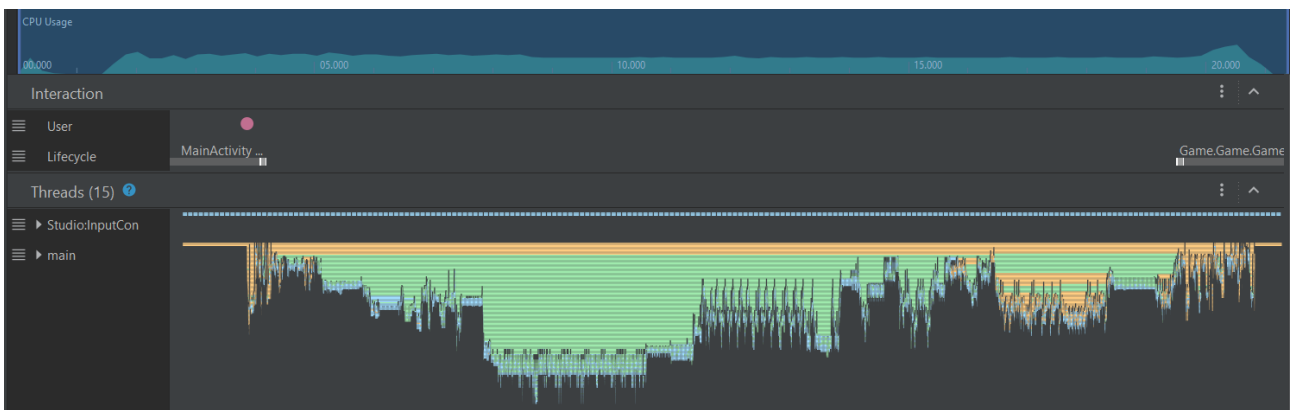


Figura 20: Emulatore: Thread "main" profilazione



In tutto questo dov'è che viene eseguito il render poiché vengono eseguite un sacco di funzioni che fanno parte del framework di Android:

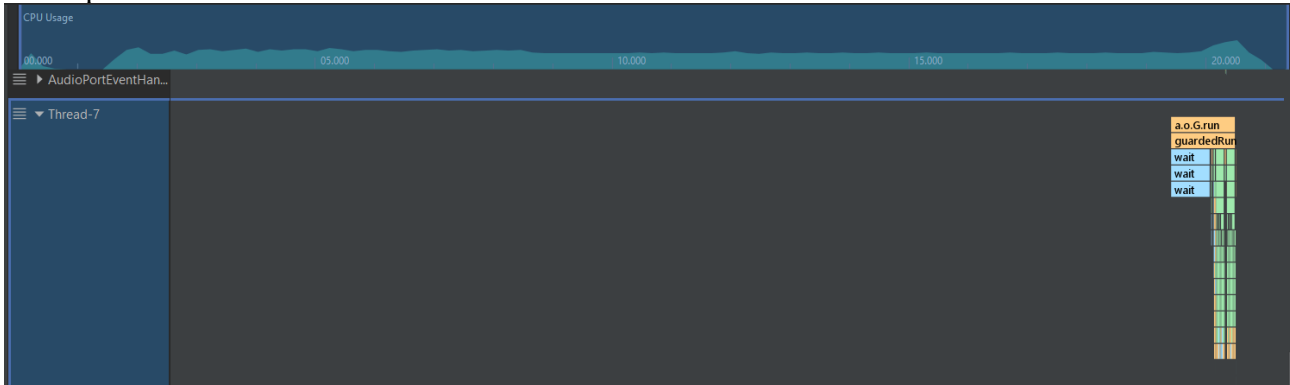


Figura 21: Emulatore: Quando viene eseguito il primo rendering

La parte dedicata al rendering da quando si avvia il gioco è molto piccola.

Questo succede solo perché c'è stato uno swap fra activity!

Tutto il tempo occupato dal Thread “main” c'è solo quando avviene lo swap fra activity.

Bisogna tenere conto di questo fattore ed analizzare l'andamento della CPU dove verrà eseguito solo il render.

Andamento della CPU nella quale chiedo di eseguire numerosi rendering concatenati:

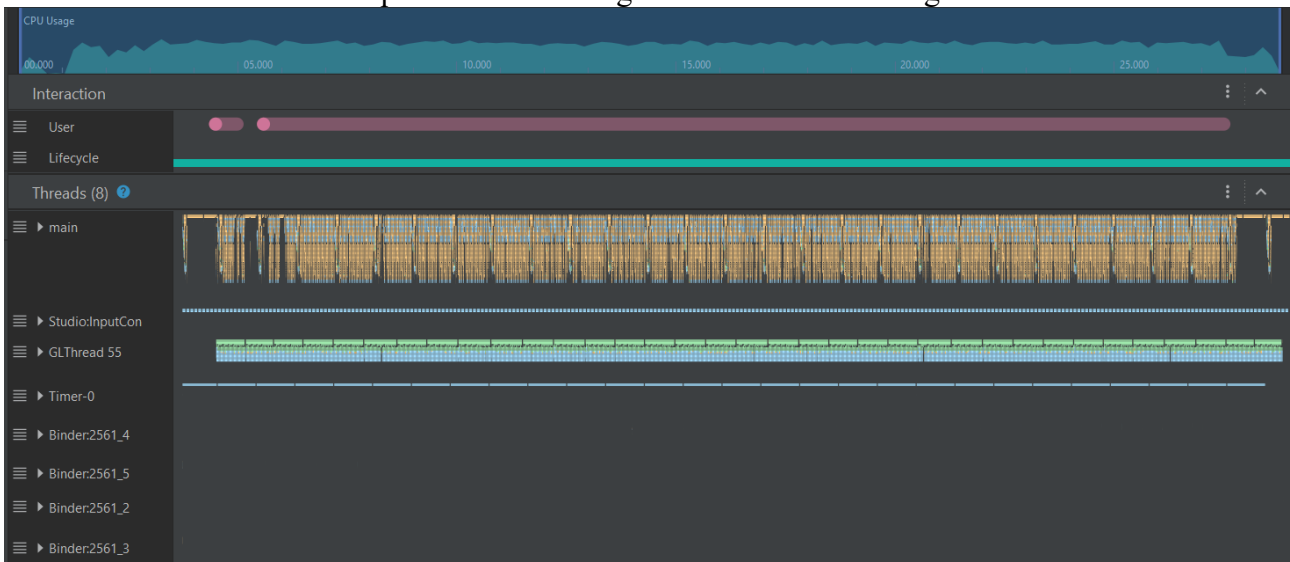


Figura 22: Emulatore: Utilizzo CPU molteplici rendering

Il Thread in cui viene eseguito il mio codice per il rendering è “GLThread 55”.

Nella riga “User” dove appare una linea rossa, quella stessa linea rossa indica l'inizio e la fine dell'evento Touch nello schermo, nel mentre è stata modificata in modo casuale la rotazione sull'asse X,Y e Z ed è stata richiesta ogni volta il rendering data la nuova rotazione ottenuta.

Dall'immagine non si nota, ma l'utilizzo tende ad essere costante al valore **25%**

a volte è superiore al 25% poiché la OpenGL per certe operazioni fa uso di un thread separato a volte l'uso scende poco sotto il 25%, in pratica l'utilizzo è 25%.

Come citato nel paragrafo “**2.1.1 Utilizzo della CPU**” questo valore dice sostanzialmente che sto **usando al massimo un Core e basta**, la bottleneck è presente nell'esecuzione di codice nella CPU più precisamente sapendo che l'utilizzo è causato dalla richiesta di rendering, si può subito dire che **la bottleneck è causata dalla funzione chiamata per eseguire un render.**

Analizzando i dati forniti dal recording:

## Flame Chart:

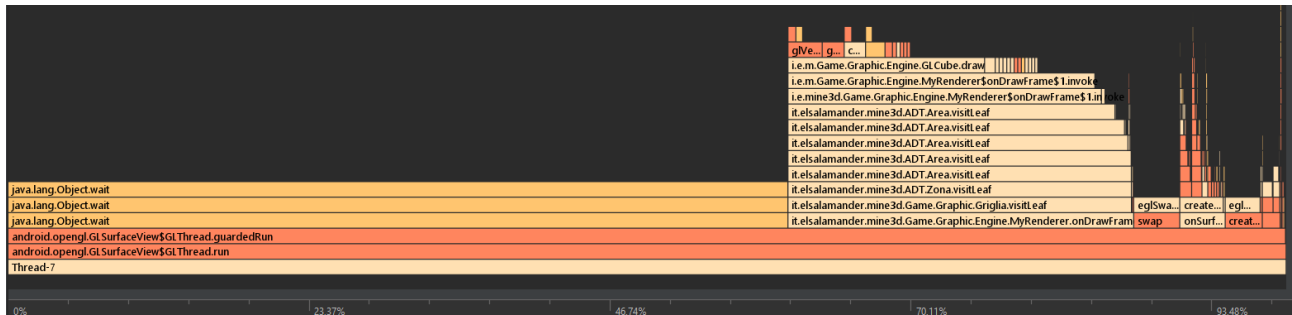


Figura 23: Emulatore: FlameChart di un render

Quello mostrato è il **Flame Chart** del primo Render fatto quando si apre il gioco.

Senza analizzare i numeri si nota già da subito un WAIT che ha una durata elevata, probabilmente un wait piazzato per via di qualche semaforo in attesa di sincronizzare qualche thread del framework, poco ci importa poiché, non posso farci niente e poi anche perché, quando vengono richiesti più rendering in cascata, non è presente nessun WAIT come si può notare dalla Figura 22, non ci sono zone morte.

La parte più interessante è data da questo pezzo:

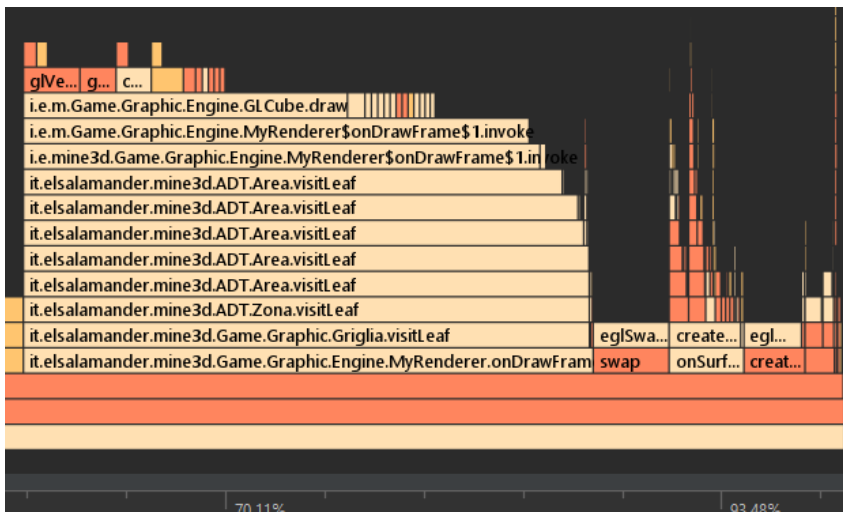


Figura 24: Emulatore: Zoom Flame Chart

Si può vedere quando viene chiamata la funzione **“onDrawFrame”** (la seconda riga gialla/arancio partendo dal basso, “it.elsalamander.mine3d.Game.Graphic.Engine.MyRenderer.onDrawFrame”)

Si nota subito un uso ricorsivo della funzione **“visitLeaf”** che porta alla terza riga dall’alto dove vengono eseguite numerose funzioni tra cui quella di nome “i.e.m.Game.Graphic.Engine.GLCube.draw” (il nome è stato accorciato per farlo stare dentro il rettangolo) in questa funzione vengono usate esclusivamente funzioni della GLES20 per completare il rendering, mentre, le altre funzioni eseguite insieme al “draw”, sono le funzioni per il calcolo per completare la matrice finale del singolo cubo, mentre le rotazioni, moltiplicazioni, copie di matrici si trovano in un punto talmente piccolo, che non si vede, sono immediatamente sopra alla funzione **“onDrawFrame”**.

**La parte preponderante è costituita dalla funzione “draw”** ma, purtroppo, in questa funzione sarà pressoché difficile migliorarla poiché, come ho già detto, vengono eseguite esclusivamente funzioni della GLES20.

Si nota inoltre, che c’è un tempo **“non segnato”** ovvero, il tempo che ci mette una funzione che chiama altre funzioni, non è uguale alla somma dei tempi delle funzioni chiamate all’interno, la

prima osservazione è data dal fatto che c'è codice e non solo chiamate ad altre funzioni, ma nel caso della funzione “draw” non è così, anche la funzione di render non ha “codice” ha quasi esclusivamente chiamate a funzioni (avrebbe alcune variabili temporanee, ma sono il risultato di somme e sottrazioni, nulla in confronto alle operazioni fra matrici), da cosa è dovuto quel tempo? Si sa che a livello macchina, la funzione non è semplice come fa sembrare la programmazione ad alto livello, di fatti bisogna salvare lo stato corrente ed eseguire la routine e poi recuperare lo stato e altro ancora, un tempo più alto di quanto mi aspettassi.

Osserviamo i numeri:

Name	Total (µs)	%	Self (µs)	%	Children (µs)	%
Thread-70_0	1,245,163	100,00	2	0,00	1,245,161	100,00
run() (android.opengl.GLSurfaceView\$GLThread)	1,245,161	100,00	203	0,02	1,244,958	99,98
guardedRun() (android.opengl.GLSurfaceView\$GLThread)	1,244,549	99,95	1,803	0,14	1,242,746	99,81
wait() (java.lang.Object)	759,841	61,02	40	0,00	759,801	61,02
onDrawFrame() (it.elsalamander.mine3d.Game.Graphic.Engine.MyRenderer)	336,864	27,05	731	0,06	336,133	27,00
swap() (android.opengl.GLSurfaceView\$EglHelper)	44,867	3,60	45	0,00	44,822	3,60
onSurfaceCreated() (it.elsalamander.mine3d.Game.Graphic.Engine.MyRenderer)	44,238	3,55	886	0,07	43,352	3,48
createSurface() (android.opengl.GLSurfaceView\$EglHelper)	36,128	2,90	83	0,01	36,045	2,89
start() (android.opengl.GLSurfaceView\$EglHelper)	17,426	1,40	220	0,02	17,206	1,38
run() (com.android.internal.view.SurfaceCallbackHelper\$1)	1,329	0,11	40	0,00	1,289	0,10
isEmpty() (java.util.ArrayList)	495	0,04	495	0,04	0	0,00
traceBegin() (android.os.Trace)	463	0,04	270	0,02	193	0,02
notifyAll() (java.lang.Object)	319	0,03	319	0,03	0	0,00
createGL() (android.opengl.GLSurfaceView\$EglHelper)	210	0,02	60	0,00	150	0,01
get() (java.lang.ref.Reference)	147	0,01	113	0,01	34	0,00
traceEnd() (android.os.Trace)	131	0,01	52	0,00	79	0,01
access\$800() (android.opengl.GLSurfaceView)	115	0,01	115	0,01	0	0,00
onSurfaceChanged() (it.elsalamander.mine3d.Game.Graphic.Engine.MyRenderer)	79	0,01	45	0,00	34	0,00
readyToDraw() (android.opengl.GLSurfaceView\$GLThread)	40	0,00	40	0,00	0	0,00
access\$1000() (android.opengl.GLSurfaceView)	29	0,00	29	0,00	0	0,00
<init>() (android.opengl.GLSurfaceView\$EglHelper)	25	0,00	17	0,00	8	0,00
append() (java.lang.StringBuilder)	128	0,01	21	0,00	107	0,01
setName() (java.lang.Thread)	113	0,01	31	0,00	82	0,01
append() (java.lang.StringBuilder)	69	0,01	16	0,00	53	0,00
<init>() (java.lang.StringBuilder)	51	0,00	21	0,00	30	0,00

Figura 25: Emulatore: Analisi tempi 1

Come commentato in precedenza, la funzione di “wait()” dura molto, ma è più interessante vedere la funzione “onDrawFrame()” che in questo caso dura **335.864 uS** (27% del totale nella finestra presa) considerazione: sono presenti 2 rendering, il tempo dedicato a un rendering è la metà, non serve a starci a pensare poiché ogni rendering ha lo stesso medesimo peso in quanto gli oggetti da renderizzare, non mutano, rimangono costanti.

onDrawFrame() (it.elsalamander.mine3d.Game.Graphic.Engine.MyRenderer)	168,678	84,34	570	0,29	168,108	84,05
visitLeaf() (it.elsalamander.mine3d.Game.Graphic.Griglia)	167,659	83,83	50	0,03	167,609	83,80
rotateM() (android.opengl.Matrix)	249	0,12	84	0,04	165	0,08
setLookAtM() (android.opengl.Matrix)	75	0,04	28	0,01	47	0,02
<init>() (it.elsalamander.mine3d.Game.Graphic.Engine.MyRenderer\$onDrawFrame\$1)	39	0,02	15	0,01	24	0,01
multiplyMM() (android.opengl.Matrix)	28	0,01	28	0,01	0	0,00
setIdentityM() (android.opengl.Matrix)	16	0,01	16	0,01	0	0,00
getGrid() (it.elsalamander.mine3d.Game.Game.Data.GameInstance)	16	0,01	16	0,01	0	0,00
arraycopy() (java.lang.System)	12	0,01	12	0,01	0	0,00
glClear() (android.opengl.GLES20)	8	0,00	8	0,00	0	0,00
getN() (it.elsalamander.mine3d.Game.Graphic.Griglia)	6	0,00	6	0,00	0	0,00
onSurfaceCreated() (it.elsalamander.mine3d.Game.Graphic.Engine.MyRenderer)	29,959	14,98	20	0,01	29,939	14,97

Figura 26: Emulatore: Analisi tempi 2

Come citato sopra, le funzioni di **moltiplicazione, rotazione e copia di matrici sono irrilevanti in confronto al tempo di esecuzione totale**, questo perché, **durante la progettazione, sono riuscito a eseguirle solo una volta, quando viene chiesto il rendering**, e non ogni volta per ogni cubo presente.

Apprendo la funzione “**visitLeaf**” fino in fondo dove si aprono le funzioni invocate si ottiene:

Name	Total (µs)	%	Self (µs)	%	Children (µs)	%
Thread-70 ()	1.245.163	100,00	2	0,00	1.245.161	100,00
run() (android.opengl.GLSurfaceView\$GLThread)	1.245.161	100,00	203	0,02	1.244.958	99,98
guardedRun() (android.opengl.GLSurfaceView\$GLThread)	1.244.549	99,95	1.803	0,14	1.242.746	99,81
wait() (java.lang.Object)	759.841	61,02	40	0,00	759.801	61,02
onDrawFrame() (it.elsalamander.mine3d.Game.Graphic.Engine.MyRenderer)	336.864	27,05	731	0,06	336.133	27,00
visitLeaf() (it.elsalamander.mine3d.Game.Graphic.Griglia)	334.686	26,88	74	0,01	334.612	26,87
visitLeaf() (it.elsalamander.mine3d.ADT.Zona)	334.597	26,87	48	0,00	334.549	26,87
visitLeaf() (it.elsalamander.mine3d.ADT.Area)	334.535	26,87	139	0,01	334.396	26,86
visitLeaf() (it.elsalamander.mine3d.ADT.Area)	334.382	26,85	826	0,07	333.556	26,79
visitLeaf() (it.elsalamander.mine3d.ADT.Area)	331.249	26,60	2.330	0,19	328.919	26,42
visitLeaf() (it.elsalamander.mine3d.ADT.Area)	327.874	26,33	6.517	0,52	321.357	25,81
visitLeaf() (it.elsalamander.mine3d.ADT.Area)	318.551	25,58	9.647	0,77	308.904	24,81
invoke() (it.elsalamander.mine3d.Game.Graphic.Engine.MyRenderer\$onDrawFrame\$1)	305.380	24,53	6.346	0,51	299.034	24,02
invoke() (it.elsalamander.mine3d.Game.Graphic.Engine.MyRenderer\$onDrawFrame\$1)	299.034	24,02	55.615	4,47	243.419	19,55
draw() (it.elsalamander.mine3d.Game.Graphic.Engine.GLCube)	191.574	15,39	72.327	5,81	119.247	9,58
getAxisValue() (it.elsalamander.mine3d.ADT.Point)	10.352	0,83	10.352	0,83	0	0,00
setDist() (it.elsalamander.mine3d.Game.Graphic.MineCube)	3.808	0,31	3.808	0,31	0	0,00
getSecond() (it.elsalamander.mine3d.util.Pair)	3.753	0,30	3.753	0,30	0	0,00
setZRend() (it.elsalamander.mine3d.Game.Graphic.MineCube)	3.718	0,30	3.718	0,30	0	0,00
getPoint() (it.elsalamander.mine3d.ADT.NodeSAH)	3.574	0,29	3.574	0,29	0	0,00
setXRend() (it.elsalamander.mine3d.Game.Graphic.MineCube)	3.556	0,29	3.556	0,29	0	0,00
scaleM() (android.opengl.Matrix)	3.541	0,28	3.541	0,28	0	0,00
translateM() (android.opengl.Matrix)	3.536	0,28	3.536	0,28	0	0,00
arraycopy() (java.lang.System)	3.429	0,28	3.429	0,28	0	0,00
getVal() (it.elsalamander.mine3d.ADT.NodeSAH)	3.408	0,27	3.408	0,27	0	0,00

Figura 27: Emulatore: Analisi tempi 3

Il peso maggiore è dovuto come già scoperto dalla funzione “**draw**” della classe **GLCube** tutte le altre funzioni eseguite sempre insieme a “**draw**” sono trascurabili.

Cosa avviene dentro la funzione “**draw**”

draw() (it.elsalamander.mine3d.Game.Graphic.Engine.GLCube)	191.574	15,39	72.327	5,81	119.247	9,58
glVertexAttribPointer() (android.opengl.GLES20)	33.383	2,68	19.084	1,53	14.299	1,15
glVertexAttribPointerBounds() (android.opengl.GLES20)	7.781	0,62	7.781	0,62	0	0,00
remaining() (java.nio.Buffer)	6.518	0,52	6.518	0,52	0	0,00
glDrawArrays() (android.opengl.GLES20)	21.407	1,72	21.407	1,72	0	0,00
checkGLError() (it.elsalamander.mine3d.Game.Graphic.Engine.GLCube)	20.807	1,67	13.396	1,08	7.411	0,60
glGetError() (android.opengl.GLES20)	7.411	0,60	7.411	0,60	0	0,00
position() (java.nio.FloatBuffer)	18.982	1,52	12.437	1,00	6.545	0,53
position() (java.nio.Buffer)	6.545	0,53	6.545	0,53	0	0,00
glEnableVertexAttribArray() (android.opengl.GLES20)	7.207	0,58	7.207	0,58	0	0,00
glUseProgram() (android.opengl.GLES20)	3.967	0,32	3.967	0,32	0	0,00
checkNotNullParameter() (kotlin.jvm.internal.Intrinsics)	3.599	0,29	3.599	0,29	0	0,00
glUniformMatrix4fv() (android.opengl.GLES20)	3.373	0,27	3.373	0,27	0	0,00
glBindTexture() (android.opengl.GLES20)	3.321	0,27	3.321	0,27	0	0,00
glActiveTexture() (android.opengl.GLES20)	3.201	0,26	3.201	0,26	0	0,00
getAxisValue() (it.elsalamander.mine3d.ADT.Point)	10.352	0,83	10.352	0,83	0	0,00

Figura 28: Emulatore: Analisi tempi 4

La funzione “**draw**” presenta la chiamata di numerose funzioni le quali più “pesanti” sono:

- **glVertexAttribPointer** 33,383 uS 17,42%
- **glDrawArrays** 21,407 uS 11,17%
- **checkGLError** 20,807 uS 10,86%
- **position** 18,982 uS 9,91%
- **resto** 31,213 uS 16,29%

(Le percentuali mostrate si riferiscono al contributo percentuale alla funzione **draw**)

Il “**resto**” è la somma dei tempi delle funzioni rimaste, qua si nota meglio quello che è stato notato guardando il Flame Chart, ovvero c’è un tempo che manca, qui in particolare manca:

$$191,574 - (33,383 + 21,407 + 20,807 + 18,982 + 31,213) = 65782 \text{ uS} \quad 34,33\%$$

Un tempo che ricopre il 34.33% della funzione ignoto, non è esattamente ignoto perché bisogna considerare quando parte il timer e quando si conclude per misurare il tempo di esecuzione di una funzione, comunque rimane un tempo purtroppo anche rilevante che contribuisce alla bottleneck.

### 5.3.2 - Profilazione dispositivo fisico

La profilazione è identica a quella fatta per l'emulatore, salterò le parti discorsive e passerò subito ai risultati significativi del recording.

La profilazione è stata eseguita con le seguenti impostazioni:

- Impostazioni Device Fisico:
  - Grandezza schermo: 6,53"
  - Risoluzione: 2340x1080, densità: xhdpi
  - Versione Android: 11 RP1A,200720.011
  - Processore: MediaTek Helio G80 octa core [Link](#)
    - 2 x 2GHz & 6 x 1.8GHz
    - DirectX 12
    - Versione OpenGL ES 3.2
    - Turbo GPU (un overclock per migliorare le prestazioni) 950 MHz
  - Ram: 4 + 1Gb LPDDR4x
  - GPU: Mali-G52 [Link](#)
    - OpenGL® ES 1.1, 2.0, 3.1, 3.2
    - Vulkan 1.0
- Impostazioni gioco:
  - Lato del cubo renderizzato è 7, ciò comporta un numero totale di cubi di 218 cubi

Utilizzo della CPU quando si avvia il gioco

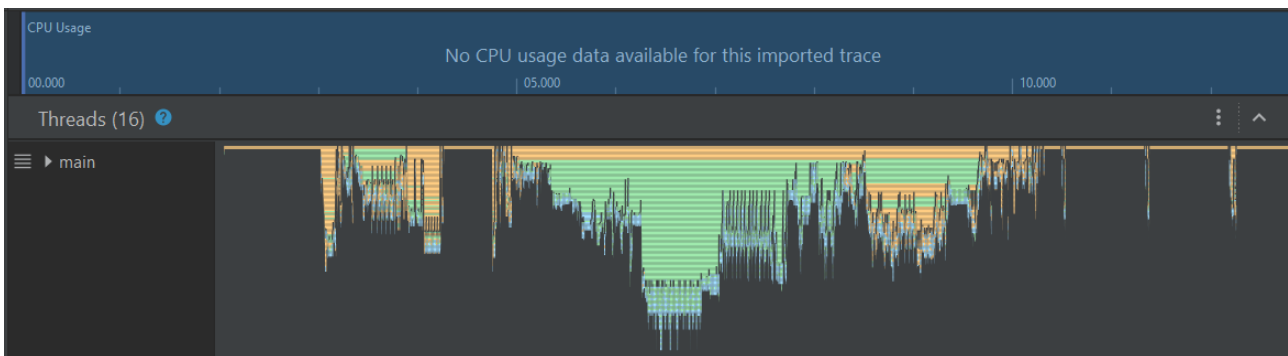


Figura 29: Device Fisico: Thread "main" profilazione

In tutto questo dov'è che viene eseguito il render poiché vengono eseguite un sacco di funzioni che fanno parte del framework di Android:

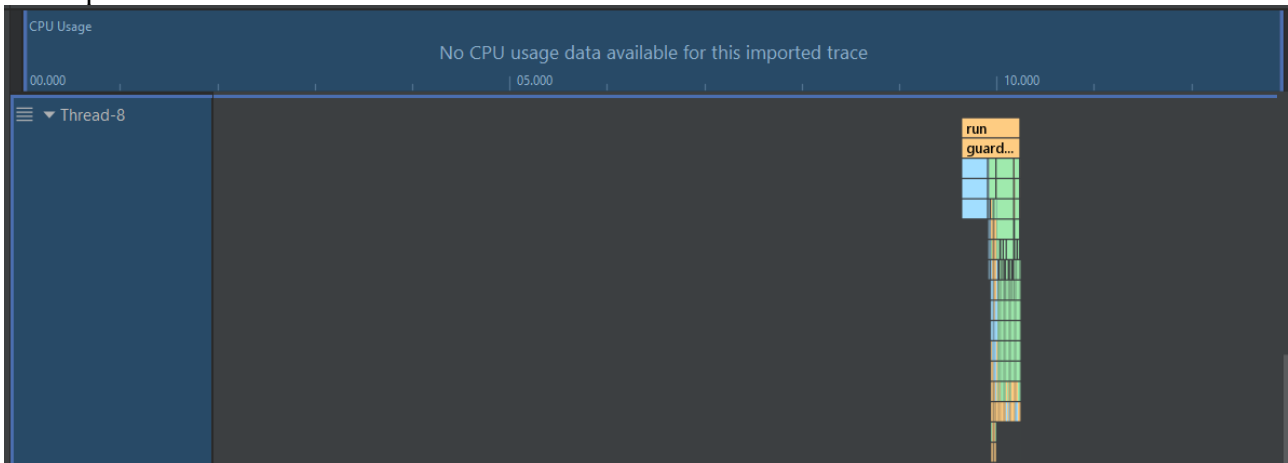


Figura 30: Device Fisico: Quando viene eseguito il primo rendering

Anche qua come nel emulatore c'è un sacco di codice che viene eseguito durante lo swap fra 2 activity.

Andamento della CPU nella quale chiedo di eseguire numerosi rendering concatenati

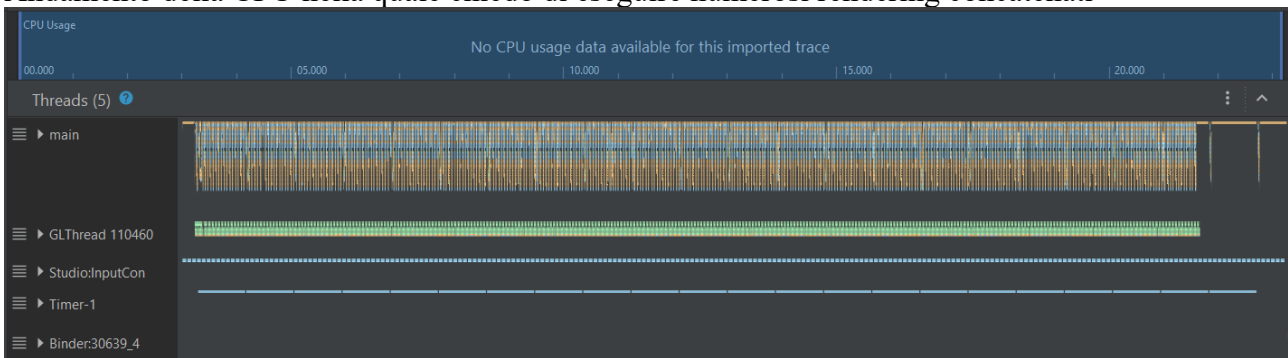


Figura 31: Device Fisico: Utilizzo CPU molteplici rendering

Un'altra immagine per vedere meglio cosa succedeva quando chiedevo molti rendering di fila:

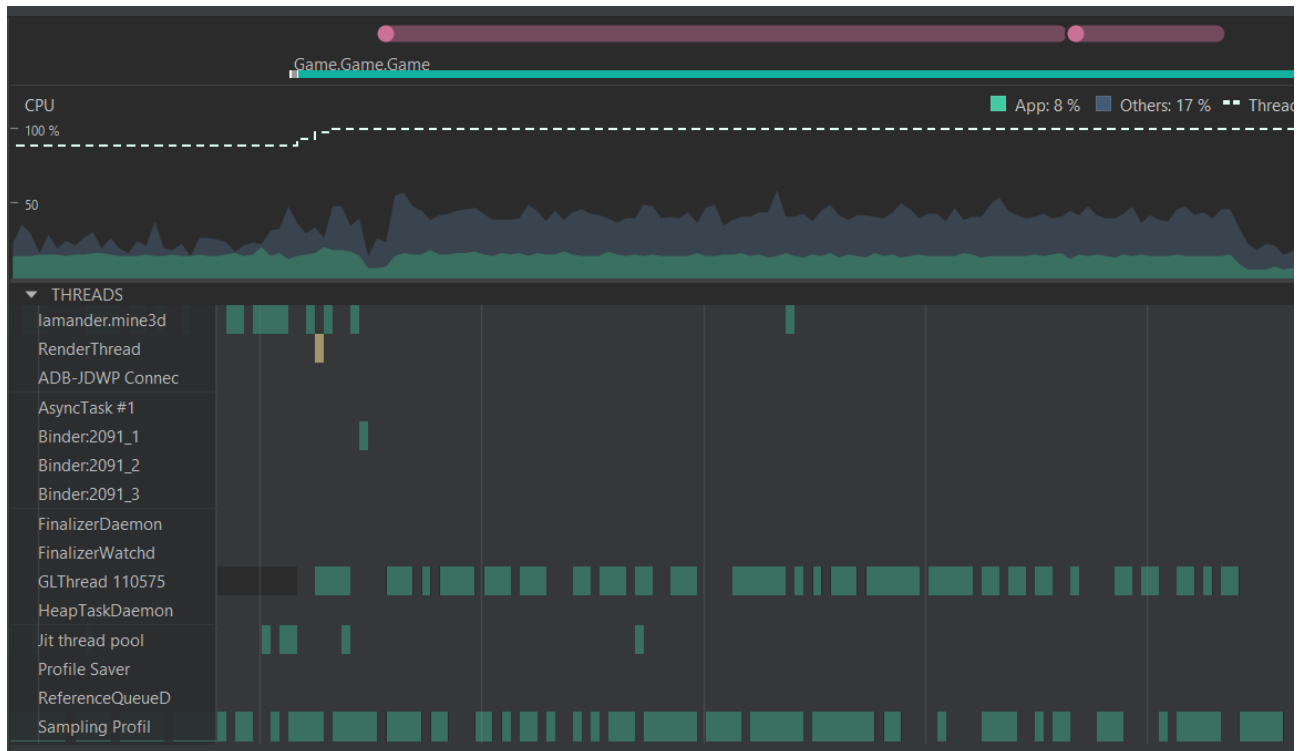


Figura 32: Device Fisico: Utilizzo CPU molteplici rendering dettaglio

Non si vede chiaramente ma, l'utilizzo della CPU sta sul 16%

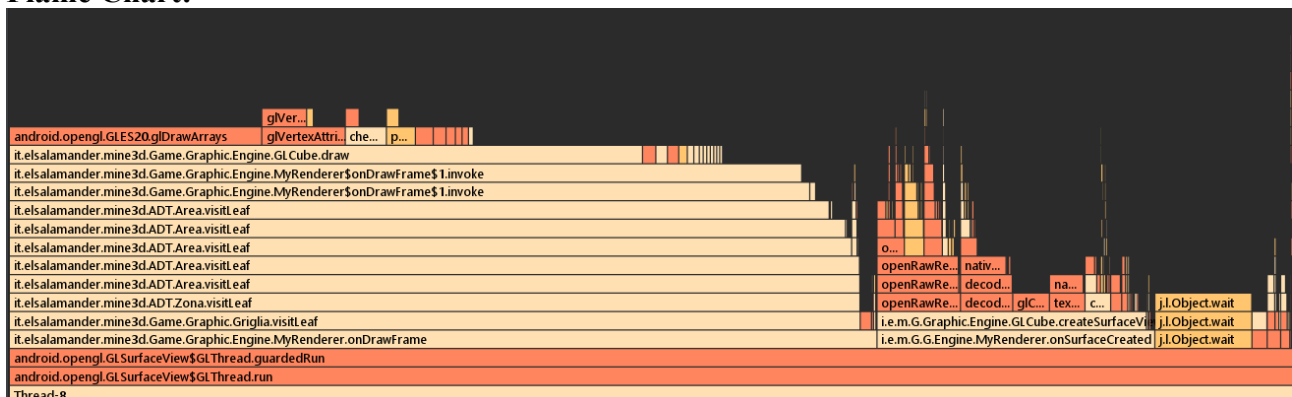
Infatti, se si considera che si fa uso del Core più veloce 2.00 GHz + Turbo GPU 950MHz, utilizzando a pieno questo core si ottieni un utilizzo del 16%

$k = 0.5$  Fattore di boost, dato che il boost non è ON/OFF ma graduale è stato scelto un valore realistico date le circostanze

$$100 * (2.00 + 0.95 * k) / (0.95 * k + 2 * 2.00 + 6 * 1.8) = 16\% \text{ circa}$$

Nota: Rispetto all'emulatore il Thread gestito dalla OpenGL "GLThread 110575" è più utilizzato.

## Flame Chart:



Nulla da commentare è identico al FlameChart dell'emulatore



## Osserviamo i numeri 1:

Name	Total (μs)	%	Self (μs)	%	Children (μs)	%
Thread-8() ()	424.454	100,00	1	0,00	424.453	100,00
run() (android.opengl.GLSurfaceView\$GLThread)	424.453	100,00	1	0,00	424.452	100,00
guardedRun() (android.opengl.GLSurfaceView\$GLThread)	424.452	100,00	622	0,15	423.830	99,85
onDrawFrame() (it.elsalamander.mine3d.Game.Graphic.Engine.MyRenderer)	286.220	67,43	656	0,15	285.564	67,28
visitLeaf() (it.elsalamander.mine3d.Game.Graphic.Griglia)	280.673	66,13	62	0,01	280.611	66,11
glClear() (android.opengl.GLES20)	3.734	0,88	3.734	0,88	0	0,00
rotateM() (android.opengl.Matrix)	668	0,16	166	0,04	502	0,12
<init>() (it.elsalamander.mine3d.Game.Graphic.Engine.MyRenderer\$onDrawFrame\$1)	132	0,03	106	0,02	26	0,01
setIdentityM() (android.opengl.Matrix)	96	0,02	96	0,02	0	0,00
multiplyMM() (android.opengl.Matrix)	93	0,02	93	0,02	0	0,00
setLookAtM() (android.opengl.Matrix)	86	0,02	27	0,01	59	0,01
getGrid() (it.elsalamander.mine3d.Game.Data.GameInstance)	45	0,01	45	0,01	0	0,00
getN() (it.elsalamander.mine3d.Game.Graphic.Griglia)	21	0,00	21	0,00	0	0,00
arraycopy() (java.lang.System)	16	0,00	16	0,00	0	0,00
onSurfaceCreated() (it.elsalamander.mine3d.Game.Graphic.Engine.MyRenderer)	91.648	21,59	1.735	0,41	89.913	21,18
wait() (java.lang.Object)	32.040	7,55	18	0,00	32.022	7,54
swap() (android.opengl.GLSurfaceView\$EglHelper)	5.026	1,18	93	0,02	4.933	1,16
start() (android.opengl.GLSurfaceView\$EglHelper)	4.441	1,05	269	0,06	4.172	0,98
createSurface() (android.opengl.GLSurfaceView\$EglHelper)	3.028	0,71	158	0,04	2.870	0,68
run() (com.android.internal.view.SurfaceCallbackHelper\$1)	505	0,12	49	0,01	456	0,11
createGL() (android.opengl.GLSurfaceView\$EglHelper)	249	0,06	100	0,02	149	0,04

Figura 33: Device Fisico: Analisi tempi 1

I tempi sono migliori rispetto all'emulatore, onestamente la cosa mi stupisce un po, poiché all'emulatore avevo dato 4 Core, in realtà il fatto che sono 4 non è molto rilevante, e ma questi erano associati a Core con un Clock di 3.30 GHz, vero anche che, l'emulatore non avrà sfruttato al massimo il Core poiché, esistono altri Thread nel Pc e anche l'emulatore deve "girare", stupito ma non troppo sorpreso a riguardo.

## Analizzando più in dettaglio

Name	Total (μs)	%	Self (μs)	%	Children (μs)	%
visitLeaf() (it.elsalamander.mine3d.ADT.Area)	277.855	65,46	1.522	0,36	276.333	65,10
visitLeaf() (it.elsalamander.mine3d.ADT.Area)	275.836	64,99	4.307	1,01	271.529	63,97
visitLeaf() (it.elsalamander.mine3d.ADT.Area)	270.452	63,72	4.468	1,05	265.984	62,66
invoke() (it.elsalamander.mine3d.Game.Graphic.Engine.MyRenderer\$onDrawFrame\$1)	264.040	62,21	2.654	0,63	261.386	61,58
invoke() (it.elsalamander.mine3d.Game.Graphic.Engine.MyRenderer\$onDrawFrame\$1)	261.386	61,58	26.099	6,15	235.287	55,43
draw() (it.elsalamander.mine3d.Game.Graphic.Engine.GLCube)	208.872	49,21	55.812	13,15	153.060	36,06
translateM() (android.opengl.Matrix)	4.389	1,03	4.389	1,03	0	0,00
getAxisValue() (it.elsalamander.mine3d.ADT.Point)	3.832	0,90	3.832	0,90	0	0,00
scaleM() (android.opengl.Matrix)	3.737	0,88	3.737	0,88	0	0,00
arraycopy() (java.lang.System)	2.825	0,67	2.825	0,67	0	0,00
setDist() (it.elsalamander.mine3d.Game.Graphic.MineCube)	2.135	0,50	2.135	0,50	0	0,00
getTextureID() (it.elsalamander.mine3d.Game.Graphic.MineCube)	1.818	0,43	1.818	0,43	0	0,00
setLarghezza() (it.elsalamander.mine3d.Game.Graphic.MineCube)	1.285	0,30	1.285	0,30	0	0,00
getPoint() (it.elsalamander.mine3d.ADT.NodeSAH)	1.174	0,28	1.174	0,28	0	0,00
getSecond() (it.elsalamander.mine3d.util.Pair)	1.174	0,28	1.174	0,28	0	0,00
setXRend() (it.elsalamander.mine3d.Game.Graphic.MineCube)	1.122	0,26	1.122	0,26	0	0,00
setYRend() (it.elsalamander.mine3d.Game.Graphic.MineCube)	988	0,23	988	0,23	0	0,00
getVal() (it.elsalamander.mine3d.ADT.NodeSAH)	984	0,23	984	0,23	0	0,00
setZRend() (it.elsalamander.mine3d.Game.Graphic.MineCube)	952	0,22	952	0,22	0	0,00
isEglia() (it.elsalamander.mine3d.ADT.NodeSAH)	1.944	0,46	1.944	0,46	0	0,00

Figura 34: Device Fisico: Analisi tempi 2

Come per l'emulatore la funzione più "pesante" quando viene eseguito il rendering è il "draw" della classe **GLCube**.

C'è da notare che alcune funzioni si sono fatte più "veloci" e altre più "lente" rispetto all'emulatore, in particolare la funzione "getAxisValue()" è circa 3 volte più veloce.



## Analizzando la funzione “draw” si ottiene:

✓ m draw() (it.elsalamander.mine3d.Game.Graphic.Engine.GLCube)	208.872	49,21	55.812	13,15	153.060	36,06
m glDrawArrays() (android.opengl.GLES20)	83.273	19,62	83.273	19,62	0	0,00
> m glVertexAttribPointer() (android.opengl.GLES20)	27.553	6,49	10.526	2,48	17.027	4,01
> m checkGLError() (it.elsalamander.mine3d.Game.Graphic.Engine.GLCube)	13.560	3,19	8.969	2,11	4.591	1,08
> m position() (java.nio.FloatBuffer)	9.567	2,25	5.399	1,27	4.168	0,98
m glUniformMatrix4fv() (android.opengl.GLES20)	5.883	1,39	5.883	1,39	0	0,00
m glEnableVertexAttribArray() (android.opengl.GLES20)	4.216	0,99	4.216	0,99	0	0,00
m glUseProgram() (android.opengl.GLES20)	3.095	0,73	3.095	0,73	0	0,00
m glBindTexture() (android.opengl.GLES20)	2.195	0,52	2.195	0,52	0	0,00
m glActiveTexture() (android.opengl.GLES20)	2.122	0,50	2.122	0,50	0	0,00
m checkNotNullParameter() (kotlin.jvm.internal.Intrinsics)	1.596	0,38	1.596	0,38	0	0,00
m translateM0() (android.opengl.Matrix)	4.389	1,03	4.389	1,03	0	0,00

Anche qua alcune funzioni sono più veloci e altre più lente, si nota che la funzione “glDrawArrays” è purtroppo circa 4 volte più lenta, mentre il resto mediamente.

La funzione “draw” sul device fisico ci mette **208,872 uS** mentre l’emulatore ci mette **191,574 uS** ma nonostante questo, il tempo per fare un rendering è **286,220 uS** per il device fisico è **336,864 uS**.

Un’altra cosa da analizzare come fatto per il device emulato è una piccola analisi del Draw Funzioni più lente:

- **glDrawArrays** **83,273 uS** **39,87%**
- **glVertexAttribPointer** **27,553 uS** **13,19%**
- **checkGLError** **13,560 uS** **6,49%**
- **posiction** **9,567 uS** **4,58%**
- **resto** **19,107 uS** **9,15%**

**208,872-(83,273 + 27,553 + 13,560 + 9,567 + 19,107) = 55,812 uS** **26,72%**

Il tempo “perso” nel dispositivo fisico è inferiore all’emulatore.

## 5.4 - Rilevazione di Bottleneck

Analizzando i risultati dei 2 dispositivi: **Fisico** e **Emulato**, si può concludere che la **bottleneck** è causata dalla **CPU**, **più precisamente dalla funzione di Rendering “draw” nella classe GLCube**. **Un altro fattore** che provoca questa bottleneck è **che l'app è multithread** per quanto riguarda la parte dedicata al rendering.

Però la cosa mi ha al quanto sorpreso e stupito, l'ultima parte di codice che avrei pensato che creasse la bottleneck era proprio la OpenGL.

Da cosa può essere causata questa cosa?

**Chiamare una funzione JNI non è diretto come sembra**, per verificarlo sono state create 2 funzioni che fanno la medesima cosa: una in kotlin e una in C++.

Questa funzione **banale**, è stata ripetuta 10.000 volte e i tempi di esecuzione della funzione scritta in C++ è circa 2.5 volte più lunga.

Questo perché, la funzione era **“banale”**, **il tempo di esecuzione del corpo della funzione in C++ era più piccolo o comparabile al tempo che viene impiegato perché avvenga la chiamata + il tempo di ritorno**.

Infatti, se si vuole ottenere dei vantaggi nel passaggio in C++ da Kotlin bisogna scrivere un corpo abbastanza lungo, per rendere trascurabile il tempo di chiamata + il tempo di ritorno, altrimenti la cosa è abbastanza inutile e in casi estremi, anche contro produttiva.

Osservato questo fatto, ovvero che **la chiamata in JNI non è veloce**, c'è da sottolineare il fatto che **la OpenGL è scritta in C++**, ma nell'app **le chiamate vengono effettuate da Kotlin**, ogni funzione usata della GLES20 chiama una funzione scritta in C++, e queste ultime sono molte. **Questo contribuisce alla bottleneck**.

## 6 Tecniche di ottimizzazione

Prima di elencare tecniche molto “esotiche” per migliorare il codice, discuto su come migliorare la bottleneck trovata, ovvero l’esecuzione della funzione “**draw**” che comprende l’uso della **JNI**.

- In modo diretto, sarebbe il caso di avere l’intera **funzione “draw” in C++**, ma non è semplice poiché tale funzione usa variabili oggetto, bisognerebbe trasferire l’intera classe in C++ e avere una classe in kotlin di interfaccia.  
Questo permetterebbe di ridurre considerevolmente le chiamate **JNI** e quindi la **riduzione di tempi dovuti alle chiamate in C++ da Kotlin**.
- Un altro aspetto della ottimizzazione è come è stata realizzata la logica, ovvero:
  - Come sono stati strutturati gli ADT usati, per avere la miglior complessità a livello computazionale e occupazione della RAM, utilizzare uno rispetto ad un altro.
  - La parte di codice che realizza la logica è composta da cicli che si possono evitare se si cambia approccio.  
(Dato che le vie “del signore” sono tante, c’è di sicuro una via più performante di quella considerata)  
Come è stato fatto, è stata applicata questa ottimizzazione a inizio progetto, come mostrato nel paragrafo 5.2.
- Un altro ritardo che è stato notato tramite i **Flame chart e poi anche a livello numerico**, anche se si faceva riferimento alla funzione “**draw**” perché più evidente, è stato notato che ci sono dei **tempi “persi”**, questo è dovuto da un fatto molto poco pensato “ed esotico”, quando si chiama una funzione ad alto livello non è semplice come a livello macchina, la parte più gravosa di una chiamata a funzione non sono i svariati PUSH e POP, che anch’essi contribuiscono, la funzione più gravosa è il **JUMP**, fra le istruzioni macchina generalmente il **JUMP è una fra le funzioni più lente fra le istruzioni base**, e non solo, se si è in presenza di processori a pipeline come nel mio caso del telefono fisico che monta una CPU che ha:
  - 2 x Arm Cortex-A75 @ 2GHz
  - 6 x Arm Cortex-A55 @ 1.8GHz

**Usa pipeline a 3 stadi, il JUMP crea bolle nella pipeline.**

Questa cosa non è a svantaggio solo delle chiamate a funzione ma a **tutti i cicli, ogni costruito che utilizza il JUMP contribuisce alla bottleneck.**

Per accorgersi di questo bisogna eseguire una quantità spropositata di **JUMP**.

Per “dimostrare” questo è stata eseguita una funzione con il corpo vuoto per 10.000.000 di volte un numero un po esagerato, questa esecuzione è stata fatta tramite un ciclo **FOR** in **Kotlin** il tempo di esecuzione è di circa **61 mS. (usando le ricorsioni invece 51 mS)**

$$0,061/10.000.000 = 6,1nS$$

$$T_{clock} = 1/(2*10^9) = 0,5 nS$$

**Per fare un ciclo del ci mette circa 12 cicli macchina, di cui in questi 12 cicli servono solo per il ciclo FOR e chiamata a funzione vuota** (che probabilmente se il compilatore è fatto bene non viene neanche eseguita la chiamata a funzione)

(Utilizzerò nei calcoli successivi il tempo di 6,1 nS perché il ciclo for è molto presente, dovrei considerare un tempo inferiore per il JUMP singolo poiché nei 12 cicli oltre al JUMP c’è anche un incremento del contatore è un comparatore)

Nel l’applicazione, più specificatamente, nel caso studiato e analizzato ci sono 218 Cubi da renderizzare, per ogni cubo ci sono circa più di 45 chiamate a funzione, per un totale circa di 10.000 chiamate a funzione, senza contare che per realizzare alcune delle funzioni chiamate

contengono altre chiamate a funzione e cicli aumentando ancora di più i **JUMP** eseguiti c'è un tempo stimato di  $10000 * 6,1 * 10^{-9} = 61 \text{ uS}$  usati per il vari **JUMP** un numero comparabile con il tempo “perso” trovato nella funzione “draw” che è di **55,812 uS**, però io sto considerando anche altre funzioni, se considero solo la funzione “draw” ho **18 chiamate a funzione per cubo quindi un totale di 3924 chiamate senza contare quelle annidate, il tempo “perso” è di:**

$$3924 * 6,1 * 10^{-9} = 24 \text{ uS}$$

**un tempo comparabile ai 55,812 uS dato che io eseguo 2 render non 1, e anche che non sto considerando che le chiamate sono in JNI.**

**Come risolvere questo problema?**

**Attributo “INLINE”** si scrive una funzione che in realtà sostituisce la chiamata con il corpo, questo permette di non far avvenire il **JUMP** però la funzione per ogni chiamata eseguita occupa uno spazio nella memoria, con questo attributo si può attenuare il ritardo dovuto dai **JUMP**.

Oltre al **inline** se si sa quante volte bisogna ripetere un funzione si può fare a meno del ciclo **FOR** questo renderà il codi meno versatile e più pesante nella memoria ma evita a livello macchina codice in più dovuto dalla realizzazione del ciclo **FOR** ma anche **WHILE** e altri.

Oltre all'inline è usata anche la **programmazione breanchless** ovvero un stile di programmazione dove si evita il più possibile l'utilizzo dei breanch.

- **Un altro modo per migliorare le prestazioni è sfruttare meglio l'hardware a disposizione**, come già mostrato il rendering usa solo un Core della CPU, per **sfruttare al meglio la presenza di più Core** bisognerebbe rendere il rendering multithread però c'è già una **limitazione data dalla OpenGL**, il quale vuole che le chiamate siano effettuate da un thread singolo.

**Il multi trading è comunque possibile in quanto si può fare un pre-rendering** in modo asincrono, ovvero quello che si può fare nell'app è creare un thread dove vengono fatti i calcoli sulle matrici e il risultato messo su una Queue, il Thread dedito al rendering finale deve prendere l'elemento nella Queue e terminare il rendering, così da **trasformare la sequenza logica data:**

**Thread A : Calcola matrice(Lento) – Renderizza(Lento)**

**nella sequenza:**

**Thread A: Calcola matrice(Lento) – immetti nella Queue(Veloce)**

**Thread B: prendi valore nella Queue(Veloce) – Termina il renderin(Lento)**

**Migliorando le prestazioni.**

C'è un semaforo involontario ma accade solo se il **Thread A** è più lento del **Thread B**, in quanto nelle analisi effettuate il calcolo della matrice è più veloce della finalizzazione del rendering il **Thread B non si ferma ad aspettare il Thread A**, invece bisogna considerare di mettere un limite alla coda prima che il **Thread A** genera un coda troppo lunga e quindi una occupazione della **RAM** inutile, basta avere pre-renderizzati 3 matrici, ma se sei vuole fare questo il **Thread A** deve essere asincrono alla chiamata di funzione per eseguire il rendering, per evitare che vengano eseguiti dei “**WAIT**” involontari nel **Thread principale** del gioco.

Questa è un tecnica molto usata anche nei motori grafici più performanti ed elaborati, più nel specifico fanno fare il **pre-rendering alla CPU per poi far terminare il rendering alla GPU.**

Questa suddivisione in 2 è supportata dal fatto che la GPU rispetto a una CPU è più lenta a fare calcoli vettoriali, anche se al giorno d'oggi puoi farlo con eccellenti librerie come OpenCL, CUDA e Thrust.

Nel caso delle schede grafiche NVIDIA ci sono acceleratori hardware chiamati CUDA per migliorare le prestazioni di questi calcoli.

- Un altro aspetto più specifico nel codice scritto nell'app bisogna limitare il codice, nel caso del “draw” era rimasta una funzione “checkGLError” che serviva per debuggare il codice se avveniva un errore questo alla fine non serve più di fatti si risparmia **13.560 uS** su un totale di **208.872 uS** si va a risparmiare circa il **6,49%** della durata del “draw”.

Sempre legato a questo fattore, c'è da notare che l'ADT realizzato, è stato studiato per un altro scopo, di conseguenza esegue codice non necessario per lo scopo qua utilizzato, ma è inutile toglierlo poiché la parte di codice viene eseguita nel PUT e REMOVE, funzioni mai eseguite durante il rendering.

- Un altro aspetto che non è da sottovalutare è la versione dell'API utilizzata in questo caso la OpenGL, inizialmente è stata usata la GLES20, si è passati alla GLES32 che ha portato a dei cambiamenti, positivi e negativi, ma tutto sommato positivi in termini di tempi di esecuzione del rendering.

Sempre parlando della OpenGL per “ottimizzare” bisognerebbe sapere quali delle funzioni di essa bisognerebbe evitare e quali invece usare al posto di altre, e ancora sapere gli infiniti attributi/impostazioni che possono essere attivati o disabilitati.

Ad esempio bisogna evitare l'uso delle funzioni glGet\* in particolare:

- Il GET delle dimensioni delle trame da OpenGL.
- Il GET delle larghezze di linea/dimensioni in punti da OpenGL.

Invece una conoscenza migliore degli attributi permette di impostare la OpenGL per eseguire il rendering più veloce a discapito della qualità e viceversa, dipende dai casi di applicazione, fra gli attributi trovati c'è

- GL\_CULL\_FACES:

Da usare “glEnable(GL\_CULL\_FACES)” imposta una texture semplice alla faccia posteriore. Questo è eccellente per ridurre il numero di poligoni.

- Sempre rimanendo nella OpenGL, una causa di rallentamenti può essere causata da texture troppo pesanti, nel progetto quando sono state create le texture è stato tenuto conto di questo fattore, infatti sono tutti quadrati di dimensioni 100x100 pixel le immagini, in modo da avere alla fine bitmap leggeri, ma con una qualità buona.

## 7 Considerazioni finali

### I numeri mostrati rispecchiano la realtà?

**No**, il profiler introduce una latenza non da poco durante l'esecuzione dell'app.

Non è invisibile alla CPU ha un peso computazionale eseguire il tracciamento delle funzioni e calcolare i tempi di esecuzione, ma questa latenza introdotta è costante per tutti i metodi, quindi le funzioni identificanti come pesanti e leggere permangono.

### E' necessario spingersi così tanto a migliorare le prestazioni?

Diciamo che "esiste" una soglia che, una volta superata, i miglioramenti non si notano più, i telefoni di solito hanno uno schermo da 60Hz, esistono anche da 144Hz ma sono device con hardware più performante.

Avere un schermo di 60Hz significa che il render deve durare al massimo 16,6 mS, scendere sotto questa soglia significa solo garantire 60Hz e di vantare un ottimo rendering, a livello utente migliorare un tempo già sotto i 16,6 mS non si nota.

## 7.1 Risultati ottimizzazione

Ottimizzazioni effettuate:

- Cambio dalla GLES20 alla GLES32
- Funzioni per calcolo delle matrici senza cicli.
- Eliminazione di funzioni ritenute inutili all'esecuzione del rendering.
- Utilizzo attributi della GLES32

Prima di ottimizzare		Dopo aver ottimizzato	
T. invocazione	T. Draw	T. invocazione	T. Draw
261,386	208,872	191,222	154,155

Il tempo dell'esecuzione dei "draw" è sceso del 26,196%

## **Sitografia:**

### **androidDeveloper**

<https://developer.android.com/agi>

<https://developer.android.com/studio/profile/android-profiler>

### **opengl**

<https://www.khronos.org/registry/OpenGL-Refpages/es3.0/>

<https://developer.android.com/guide/topics/graphics/opengl>

<https://developer.android.com/training/graphics/opengl/projection>

### **Eventi di tocco**

<https://developer.android.com/training/gestures>

### **Per i file JSON**

<https://support.smartbear.com/alertsite/docs/monitors/api/endpoint/jsonpath.html>

### **Per gli eventi del gioco**

<https://kotlinlang.org/docs/annotations.html>

<https://kotlinlang.org/docs/reflection.html#function-references>

<https://discuss.kotlinlang.org/t/how-to-declare-a-generic-key-value-variable-with-class-types-as-key/13364>

### **Discussioni varie di problemi**

<https://stackoverflow.com/>

### **ADT**

<https://worldoftanks.eu/en/news/general-news/ray-tracing/>

[https://en.wikipedia.org/wiki/Bounding\\_volume\\_hierarchy](https://en.wikipedia.org/wiki/Bounding_volume_hierarchy)

[https://it.wikipedia.org/wiki/Albero\\_quadramentale](https://it.wikipedia.org/wiki/Albero_quadramentale)