# POLITECNICO DI TORINO

Collegio di Ingegneria Informatica, del Cinema e Meccatronica

**Corso di Laurea Magistrale**
**in Ingegneria del Cinema e dei Mezzi di Comunicazione**


Tesi di Laurea Magistrale

# Ricerca e Sviluppo su Game Engine per il Deploy su Web di Videogiochi



**Relatori**

*firma del relatore*
prof. Marco Mazzaglia
*firma del correlatore*
prof. Francesco Strada

**Candidato**

*firma del candidato*
Michele Bissanti


Ottobre 2025

# Abstract

Lo sviluppo di applicazioni grafiche 3d o di videogiochi che girano sul web è un argomento di crescente rilevanza e sta diventando sempre più una sfida importante per le aziende del settore. La maturazione di tecnologie come WebGL e WebGPU permette ora di ottenere prestazioni sempre migliori, ma la scelta cruciale rimane comunque quella del motore di gioco utile poi ad esportare il prodotto con le tecnologie sopra citate.

In questa tesi il problema viene affrontato attraverso un'analisi comparativa di quattro motori di gioco (Unity, Godot, PlayCanvas e Rogue Engine) scelti per rappresentare sia gli engine nativi che consentono l'esportazione in WebGL/WebGPU, sia quelli progettati nativamente per il web.

L'obiettivo principale è comprendere in che modo le differenze di export tra queste piattaforme influenzino le prestazioni effettive e la qualità complessiva dell'esperienza utente durante l'esecuzione di applicazioni 3D nel browser.

Per la sperimentazione è stato sviluppato un prototipo di endless runner 3D, denominato Knight Runner, mantenendo la stessa complessità di scena e logica di gioco in tutti gli engine. I test sono stati condotti su un insieme eterogeneo di dispositivi e browser raccogliendo dati quantitativi e qualitativi relativi a FPS medi, frame time, deviazione standard, First Load Time e compatibilità cross-device.

I risultati mostrano che Unity risulta il motore con le prestazioni più elevate, pur presentando una leggera instabilità nel frame rate, specialmente su dispositivi mobili. Godot, al contrario, offre una maggiore stabilità dei frame ma con prestazioni complessivamente inferiori e tempi di caricamento più lunghi. PlayCanvas si distingue per l'efficienza delle build, con dimensioni ridotte e ottima compatibilità cross-browser, a fronte di alcune limitazioni nella fisica e nella gestione avanzata delle animazioni. Rogue Engine, ancora in fase di maturazione, si è rivelato poco stabile e limitato dal punto di vista editoriale e documentale, risultando meno adatto a progetti complessi o di produzione industriale.

Le analisi confermano le ipotesi formulate: gli engine nativi tendono a garantire una maggiore completezza di strumenti e un workflow più robusto, ma con un evidente gap prestazionale nelle build web, mentre gli engine web-native offrono maggiore leggerezza e compatibilità a scapito delle funzionalità avanzate.

Dal punto di vista applicativo, lo studio fornisce un quadro di riferimento utile per la scelta del motore di gioco in progetti destinati al web, evidenziando che Unity rappresenta attualmente la soluzione più affidabile per produzioni professionali, mentre PlayCanvas costituisce un'alternativa promettente per esperienze leggere e ad alta portabilità.

# Sommario

# 1. Introduzione

## 1.1 Contesto e motivazioni

Lo sviluppo di applicazioni interattive complesse, in particolare i videogiochi, ha subito una forte trasformazione grazie alla crescente adozione del web come piattaforma di distribuzione. Tecnologie come HTML5, WebGL e WebGpu hanno reso possibile offrire sul browser esperienze grafiche e ludiche sempre più avanzate, senza la necessità di plugin o installazioni aggiuntive. Questa transizione non riguarda solo il mercato dei videogiochi tradizionali, ma coinvolge anche settori come l'istruzione, la comunicazione e la realtà virtuale e aumentata, in cui la possibilità di fruire di contenuti interattivi su diverse piattaforme rappresenta un vantaggio competitivo significativo.

La realizzazione di videogiochi e applicazioni per il web presenta una serie di sfide legate alle differenti implementazioni offerte dai motori di gioco disponibili. Gli engine consolidati, come Unity o Unreal Engine, offrono strumenti avanzati, ma presentano delle limitazioni quando il target è il rilascio su browser. Al contrario, le soluzioni più recenti, nate e native per il web, come PlayCanvas, Babylon.js, Three.js o Rogue Engine, garantiscono una maggiore integrazione con l'ecosistema HTML5, a scapito, talvolta, di funzionalità avanzate o della maturità dell'ecosistema di sviluppo. Comprendere i punti di forza e le criticità di ciascun approccio è dunque essenziale per orientare le scelte progettuali e produttive.

La motivazione che ha generato questo progetto deriva da due esigenze distinte. Da un lato, la componente scientifica, finalizzata a colmare la carenza di studi comparativi sistematici tra engine nativi e web-native in contesti applicativi reali, mediante lo sviluppo di prototipi tridimensionali e la definizione di un set di metriche per la valutazione delle performance. Dall'altra parte, la pratica e l'approccio industriale, correlato alla collaborazione con l'azienda Tiny Bull Studios, operante nello sviluppo di videogiochi e applicazioni interattive per il web. I risultati di questa ricerca, pertanto, rivestono un valore non solo accademico, ma anche pratico, in quanto forniranno all'azienda uno strumento concreto per ottimizzare le proprie pipeline di sviluppo e offrire ai clienti soluzioni più performanti e sostenibili.

## 1.2 Obiettivi della ricerca

L'obiettivo principale di questa tesi è quello di analizzare e confrontare diversi motori di gioco nativi e web-native in scenari di deploy su browser, con particolare attenzione agli aspetti di performance, compatibilità e usabilità. A tal fine verranno sviluppati prototipi tridimensionali, che consentiranno di verificare in condizioni controllate le prestazioni delle principali tecnologie oggi disponibili.

La ricerca si articola in più obiettivi specifici:

- Definire una metodologia di benchmark che includa metriche significative per il contesto web, quali: frame rate medio massimo e minimo, tempo di caricamento e compatibilità cross-browser e cross-device.
- Valutare le differenze prestazionali tra motori di gioco nativi con supporto WebGL/WebGpu (es. Unity WebGL, Godot HTML5) e motori progettati specificamente per il web (es. PlayCanvas, Babylon.js, Three.js, Rogue Engine).
- Identificare punti di forza e criticità di ciascun engine sia dal punto di vista tecnico (prestazioni, funzionalità disponibili, limiti architetturali) che dal punto di vista pratico (curva di apprendimento, workflow di sviluppo, supporto della community).

- Offrire un quadro comparativo strutturato che possa essere di riferimento per aziende e sviluppatori interessati al deploy di videogiochi e applicazioni interattive sul web.

Accanto a questi obiettivi di natura accademica, il lavoro si propone anche di generare un impatto concreto in ambito industriale. In collaborazione con Tiny Bull Studios, i risultati ottenuti potranno essere applicati a processi reali di sviluppo e contribuiranno a orientare le scelte tecnologiche dell'azienda, con l'obiettivo di ridurre tempi e costi di produzione e di garantire esperienze di gioco più performanti e accessibili.

## 1.3 Domande di ricerca e ipotesi

Le domande di ricerca che guidano questo lavoro possono essere sintetizzate come segue:

- D1: Quali sono le prestazioni effettive dei principali motori di gioco quando utilizzati per il deploy di applicazioni 3D sul web?
- D2: In che misura esistono differenze tra engine nativi con supporto WebGL/ WebGPU (ad esempio Unity, Godot) e engine nati per il web (come PlayCanvas, Babylon.js, Three.js, Rogue Engine)?
- D3: Quali criteri possono orientare la scelta di un motore di gioco in base al contesto applicativo e ai requisiti del progetto?

A partire da queste domande si formulano le seguenti ipotesi:

- R1: I motori nativi, pur garantendo strumenti di sviluppo più completi, presentano un gap prestazionale quando esportati per il web, soprattutto in termini di tempi di caricamento.
- R2: I motori web-native, progettati per integrarsi con HTML5 e WebGL, offrono migliori tempi di avvio e compatibilità cross-browser, ma con limitazioni sul piano delle funzionalità avanzate.
- R3: Non esiste un engine "migliore" in senso assoluto, ma le prestazioni e l'efficienza dipendono fortemente dal tipo di progetto, dalla complessità della scena e dal dispositivo target.

## 1.4 Metodologia

Per rispondere alle domande di ricerca e verificare le ipotesi formulate, è stata definita una metodologia articolata nelle seguenti fasi:

1. Revisione della letteratura: analisi dei contributi scientifici e tecnici più rilevanti sul tema dei game engine per il web, delle tecnologie WebGL e WebGPU, e dei benchmark esistenti.
2. Selezione dei motori di gioco: scelta di un insieme di engine rappresentativi, comprendente sia soluzioni consolidate con export web (Unity, Godot) sia motori web-native (PlayCanvas, Three.js, Rogue Engine).
3. Sviluppo di prototipi 3D: implementazione di uno gioco dimostrativo realizzato con ciascun engine, mantenendo coerenza in termini di asset, logica di gioco e livello di complessità.
4. Definizione delle metriche di valutazione: individuazione dei parametri chiave da monitorare (FPS medio, massimo e minimo, tempo di caricamento, compatibilità cross-browser e cross-device).

5. Esecuzione dei test: deploy dei prototipi su diversi browser e dispositivi, raccolta dei dati sperimentali e analisi quantitativa delle prestazioni.
6. Analisi comparativa: confronto dei risultati ottenuti tra i diversi engine, identificazione delle aree di forza e debolezza.
7. Sintesi dei risultati: formulazione di linee guida e raccomandazioni per la scelta del motore più adatto in funzione delle esigenze progettuali.

## 1.5 Struttura della tesi

La tesi è organizzata come segue:

- Capitolo 1 – Introduzione: presenta il contesto della ricerca, le motivazioni alla base dello studio, gli obiettivi perseguiti, le domande di ricerca, la metodologia adottata e la struttura complessiva del lavoro.
- Capitolo 2 – Stato dell'arte: analizza l'evoluzione delle tecnologie per videogiochi web, descrive i principali motori disponibili e sintetizza i contributi presenti in letteratura sui benchmark e sulle prestazioni degli engine.
- Capitolo 3 – Analisi degli engine: descrive i motori selezionati per l'indagine, evidenziandone caratteristiche, architettura, funzionalità e limiti.
- Capitolo 4 – Metodologia sperimentale: illustra nel dettaglio il protocollo di benchmark adottato, il prototipo sviluppato, le metriche di riferimento e gli strumenti di misurazione utilizzati.
- Capitolo 5 – Implementazione: documenta il processo di sviluppo dei prototipi nei diversi engine e le scelte progettuali adottate.
- Capitolo 6 – Risultati: presenta i dati raccolti dai benchmark e ne propone un'analisi comparativa tra engine, browser e dispositivi.
- Capitolo 7 – Discussione dei risultati raggiungi ed eventuali comparazioni.
- Capitolo 8 – Conclusioni e sviluppi futuri: sintetizza i principali contributi della ricerca, ne evidenzia i limiti e suggerisce possibili estensioni e direzioni future.

# 2. Stato dell'arte

## 2.1 Evoluzione delle tecnologie per videogiochi web

Il panorama del gaming basato sul web ha attraversato una lunga trasformazione, che va dai primi plugin proprietari fino alle tecnologie moderne come WebGL, WebAssembly e WebXR. Questo passaggio ha radicalmente cambiato le modalità di sviluppo, distribuzione e fruizione di giochi e applicazioni interattive nel browser, rendendo necessario un approfondimento accurato delle tappe principali.

### 2.1.1 Dal plugin (Flash, Unity Web Player) al WebGL/WebAssembly

Le prime esperienze di videogiochi nel browser hanno fatto uso di plugin come Java Applets, Shockwave e soprattutto Flash, grazie alla sua facilità d'uso e alla diffusione capillare su browser (oltre il 98 % nel 2002) [1]. Flash ha permesso un breve periodo di grande diffusione e creatività nel gaming online, ma ha mostrato limiti evidenti, in particolare legati alla sicurezza, al supporto mobile e alle performance.

A partire dal tardo 2010, iniziò la transizione verso il paradigma HTML5 + JavaScript, e in seguito verso le API grafiche native del browser: WebGL (Web Graphics Library), standardizzato dal Khronos Group nel 2011, che ha portato rendering 2D e 3D accelerato via GPU nativamente nel browser, eliminando la dipendenza da plugin esterni.

Parallelamente, WebAssembly ha iniziato ad affermarsi nel 2015 come linguaggio intermedio a basso livello, compilabile da C/C++ (e altri linguaggi) e in grado di raggiungere performance vicine al codice nativo. Studi empirici su WebAssembly mettono in luce che le applicazioni compilate da benchmark scientifici risultano solo circa il 10% più lente rispetto al codice nativo, mentre analisi più approfondite su applicazioni complesse indicano possibili rallentamenti fino a 2.5×, a seconda della complessità del runtime e delle API disponibili.[2] Nel contesto della realtà virtuale e aumentata, l'uso di WebAssembly promette interoperabilità a basse latenze su dispositivi AR/VR, favorendo uno sviluppo "write-once-deploy-everywhere".[3]


*Figura 1 - Fonte: https://blog.logrocket.com/webassembly-how-and-why-559b7f96cd71/*

### 2.1.2 Standard e API moderne (WebGL, WebGPU, WebXR)

L'evoluzione prosegue con le nuove API e standard orientati a offrire funzionalità più avanzate, controllo sul rendering e compatibilità con dispositivi AR/VR.

---

[1] Mehanna e Rudametkin, «Caught in the Game».
[2] Jangda et al., «Not So Fast».
[3] Kim e Khomtchouk, «WebAssembly enables low latency interoperable augmented and virtual reality software».

WebGL rimane oggi la base consolidata per il rendering grafico web, ma si sta progressivamente affiancando o sostituendo con WebGPU. Quest'ultimo, standardizzato come "Candidate Recommendation" nel luglio 2025, fornisce accesso più esplicito e moderno alla GPU, ispirandosi a Vulkan, Metal e Direct3D 12, con minore overhead CPU e supporto per calcoli paralleli avanzati.[4] Attualmente WebGPU è già disponibile in Chrome e Edge dalla versione 113 (o tramite flag), mentre Firefox e Safari lo stanno implementando in modo crescente.

Parallelamente, il consorzio W3C ha promosso il WebXR Device API, con l'obiettivo di standardizzare l'accesso a dispositivi di realtà virtuale e aumentata direttamente nel browser. WebXR supera WebVR, offrendo un set più completo di funzionalità (tracciamento testa, controller, rendering stereoscopico, hit-test, overlay DOM, input hand tracking, ecc.)[5].

**Riepilogo Evolutivo**

| EPOCA | TECNOLOGIA / API | CARATTERISTICHE PRINCIPALI |
|---|---|---|
| **PLUGIN (ANNI '90–2000)** | Flash, Java Applets | Ampia diffusione, ma plugin proprietari; limitazioni su mobile/sicurezza |
| **STANDARD WEB INIZIALI** | HTML5 + Canvas + JavaScript | Animazioni e interazione senza plugin |
| **WEBGL (2011)** | API grafica GPU nativa | Rendering 2D/3D accelerato via browser |
| **WEBASSEMBLY (DAL 2015)** | Bytecode ad alte prestazioni | Vicino al nativo; interoperabile; utile anche per AR/VR |
| **WEBGPU (2021+)** | GPU moderno + compute shaders | API esplicita, performance avanzate, control espanso |
| **WEBXR (2022+)** | XR Device API | VR/AR via Web; tracciamento, controller, interoperabilità |

Con questa panoramica sull'evoluzione tecnologica, possiamo ora passare a delineare i criteri di confronto dei motori di gioco e come interpretare il loro supporto e performance, alla luce di questi standard.

[4] «WebGPU».
[5] «WebXR Device API».

## 2.2 Tipologie di engine per il web

Il panorama attuale degli engine per videogiochi web si può suddividere in due grandi categorie: i motori nativi con supporto per l'export web e i motori nati direttamente per il browser. Questa distinzione è importante perché riflette due approcci progettuali profondamente diversi: da un lato l'adattamento di ecosistemi già consolidati (con tutti i vantaggi e i limiti che ne conseguono), dall'altro lo sviluppo di soluzioni concepite fin dall'inizio per sfruttare le tecnologie HTML5 e WebGL/WebAssembly.

### 2.2.1 Engine nativi con export web (Unity, Godot, Unreal)

Unity è probabilmente l'esempio più noto di engine nativo che supporta il deploy su web. A partire dalla dismissione del Unity Web Player (basato su plugin NPAPI), Unity ha introdotto il supporto WebGL tramite Emscripten e il compilatore IL2CPP, che converte il codice C# in C++ e successivamente in WebAssembly/asm.js. Tuttavia, diversi studi dimostrano che le build WebGL di Unity presentano performance inferiori rispetto alle controparti native, in particolare per quanto riguarda tempi di caricamento, gestione della memoria e mancanza di multithreading completo. Ad esempio, Berdak e Plechawska-Wójcik (2017) evidenziano come le limitazioni di WebGL e la dipendenza dal single-threading impattino significativamente sul frame rate e sulla reattività delle applicazioni.[6]


*Figura 2 - Interfaccia di Unity - fonte: Unity.com*

Godot rappresenta un'alternativa open source che, a partire dalla versione 3.x, offre export in HTML5 tramite compilazione con Emscripten. Rispetto a Unity, l'approccio di Godot è più leggero, ma la maturità del supporto WebGL è ancora inferiore. Tufegdžić et al. (2024) sottolineano come l'integrazione con WebAssembly permetta a Godot di ottenere prestazioni accettabili per prototipi e applicazioni didattiche, ma con limitazioni evidenti nel supporto a feature avanzate come la fisica 3D complessa o l'uso intensivo di shader. [7]

[6] Berdak e Plechawska-Wójcik, «Performance Analysis of Unity3D Engine in the Context of Applications Run in Web Browsers».
[7] Tufegdžić et al., «Application of WebAssembly Technology in High-Performance Web Applications».

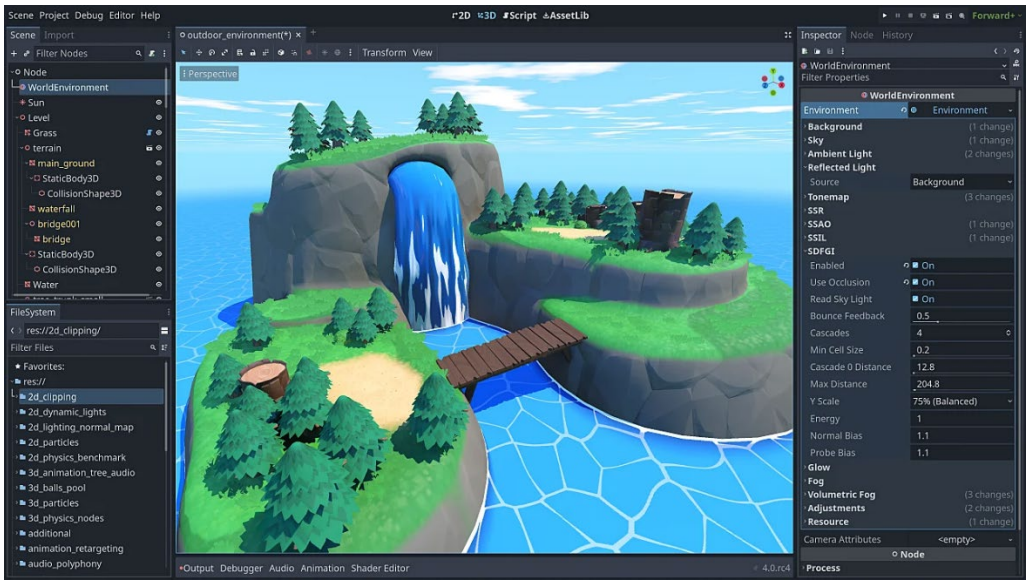



*Figura 3 - Interfaccia di Godot - Fonte: godotengine.org*

Unreal Engine, pur avendo introdotto in passato un export per HTML5, ha progressivamente ridotto il supporto ufficiale a partire dalla versione 4.24, lasciando la manutenzione alla community. Sempre le community online riportano che, sebbene l'engine sia estremamente potente, la complessità delle build HTML5 e le dimensioni elevate dei pacchetti lo rendono poco adatto a scenari di produzione orientati al web. [8]

In generale, gli engine nativi con export web offrono un ecosistema ricco di strumenti (editor avanzati, pipeline consolidate, asset store), ma presentano colli di bottiglia significativi legati a:

- tempi di caricamento elevati,
- consumo di memoria superiore rispetto agli engine nati per il web,
- compatibilità limitata su alcuni browser e dispositivi mobili.

### 2.2.2 Engine web-native (PlayCanvas, Phaser, Babylon.js, Three.js, Rogue Engine)

Accanto agli engine nativi con export, negli ultimi anni si sono diffusi motori web-native, progettati fin dall'inizio per funzionare nel browser senza passaggi intermedi di compilazione. Questi engine si basano su JavaScript/TypeScript e sfruttano direttamente WebGL (e più recentemente WebGPU).

PlayCanvas è uno dei motori web-native più noti, acquisito da ARM nel 2017. Offre un editor visuale completamente online, supporto per la collaborazione in tempo reale e un runtime altamente ottimizzato per WebGL. Secondo Halsas (2017), PlayCanvas è tra gli engine che meglio bilanciano performance e facilità di deploy, risultando adatto a progetti interattivi commerciali e applicazioni 3D complesse. [9]

[8] «Unreal Engine 4.23 released!»
[9] Halsas, *Comparison of HTML5 Game Engines Used in Game Development*.

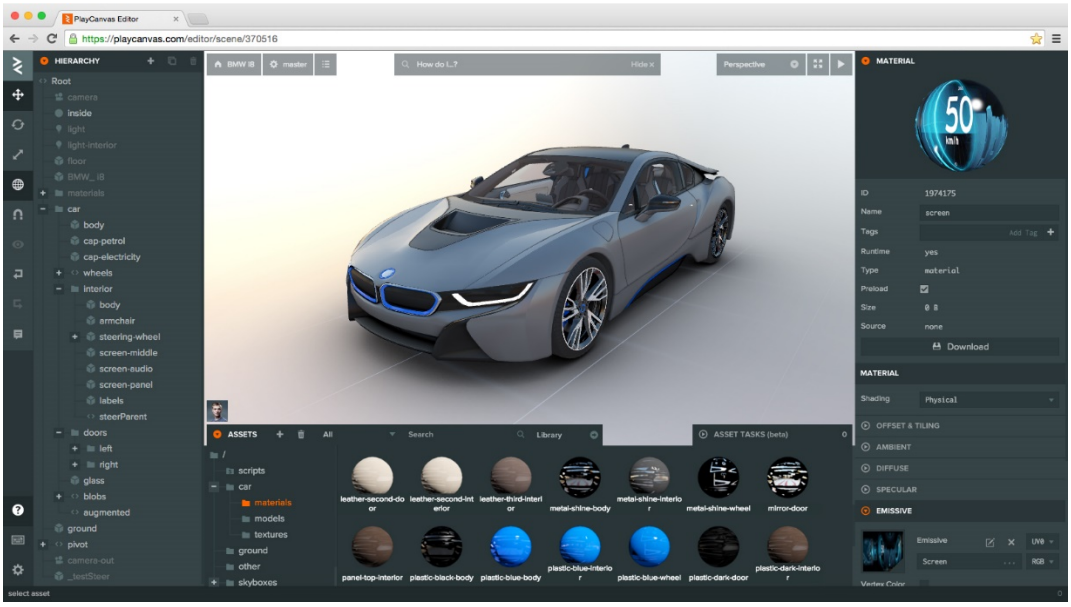



*Figura 4 - Interfaccia di Playcanvas - fonte: wikipedia.org*

Phaser, invece, è più orientato allo sviluppo di giochi 2D. Grazie alla semplicità dell'API e alla vasta community, è molto utilizzato in ambito didattico e indie. Tuttavia, la mancanza di un supporto nativo al 3D lo rende meno adatto al confronto diretto con engine general purpose.

Babylon.js e Three.js sono due framework JavaScript per lo sviluppo di applicazioni 3D. Entrambi sfruttano WebGL come backend grafico, ma con filosofie diverse:

- Three.js è una libreria più leggera e modulare, che fornisce strumenti di base per la grafica 3D e lascia grande libertà allo sviluppatore.
- Babylon.js è un framework più completo, che integra un motore fisico, gestione delle scene, materiali avanzati e supporto diretto a WebXR.

Uno studio comparativo di Johansson (2021) mostra come, a parità di scena, le performance in termini di frame rate siano simili, ma Babylon.js presenti un consumo di memoria sensibilmente superiore a Three.js. [10]

Infine, Rogue Engine è un progetto recente costruito sopra Three.js, che aggiunge un layer architetturale in stile ECS (Entity-Component-System) e un editor visuale. L'obiettivo è colmare il divario tra la flessibilità di Three.js e la necessità di strumenti integrati tipici di engine come Unity. Anche se meno maturo, rappresenta un interessante esempio di evoluzione degli engine web-native, con un focus sul workflow di sviluppo piuttosto che solo sulle performance.

In generale, gli engine web-native mostrano vantaggi significativi in termini di:

- tempi di caricamento più rapidi, grazie a bundle leggeri e modularità,
- integrazione nativa con lo stack web, che facilita la distribuzione,
- supporto immediato a WebXR, che li rende particolarmente adatti a esperienze immersive cross-platform.

[10] Johansson, *Performance and Ease of Use in 3D on the Web*.

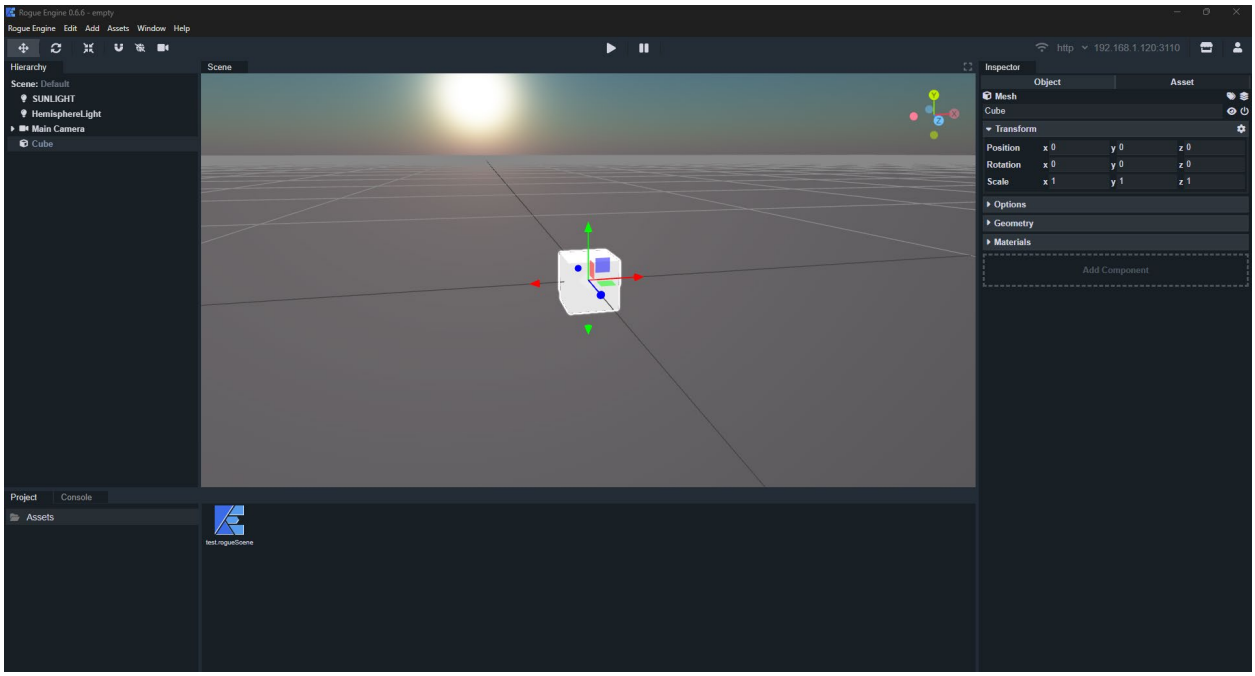



*Figura 5 - Interfaccia di Rogue Engine*

## 2.3 Studi comparativi e benchmark esistenti

La letteratura scientifica e tecnica ha analizzato in più occasioni le prestazioni dei motori di gioco in contesti web e mobile, con approcci e metriche differenti. Questa sezione sintetizza i principali risultati emersi dai lavori presi in esame, con particolare attenzione agli strumenti di valutazione e alle metodologie sperimentali utilizzate.

### 2.3.1 Metriche più utilizzate (FPS, FLT, memoria, CPU/GPU, compatibilità)

Gli studi comparativi sugli engine convergono su un set relativamente stabile di metriche di valutazione:

- Frame per secondo (FPS): è la metrica più utilizzata per misurare la fluidità del rendering. Barczak e Woźniak (2019) hanno utilizzato l'FPS medio come parametro chiave per confrontare Unity, Unreal e CryEngine . Analogamente, Pattrasitidecha (2014) ha impiegato l'FPS per valutare le prestazioni di Unity3D e ShiVa3D in ambiente mobile. [11] [12]
- First Load Time (FLT): conosciuto anche in letteratura come Startup Time o Time to First Frame, misura il tempo necessario al completamento del caricamento iniziale fino alla visualizzazione del primo frame giocabile o interattivo. Questa metrica è particolarmente rilevante nel contesto delle applicazioni web-based, dove il caricamento di asset e librerie può incidere sensibilmente sull'esperienza dell'utente finale. Studi recenti (ad esempio Bi et al., 2023; Tufegdžić et al., 2024) hanno sottolineato come il FLT rappresenti un parametro critico per valutare l'efficienza del deployment WebGL o WebGPU, oltre che per identificare colli di bottiglia nella fase di inizializzazione.
- Frame Latency Time: meno diffuso ma presente in studi recenti, misura la latenza tra input e rendering del frame. Tufegdžić et al. (2024) hanno utilizzato FLT come indicatore per testare la reattività di applicazioni ad alte prestazioni in WebAssembly. [13]
- Memoria allocata e gestione delle risorse: in diversi lavori è stato monitorato l'uso della memoria RAM, spesso tramite profiler integrati negli engine. Berdak e Plechawska-Wójcik (2017) hanno osservato che le build WebGL di Unity consumano più memoria rispetto alle controparti native. [14]
- Carico su CPU e GPU: è stato considerato soprattutto negli studi legati a WebAssembly e WebGL. In particolare, Bi et al. (2023) hanno evidenziato l'impatto del rendering e dell'ottimizzazione GPU nelle esperienze di realtà estesa via browser. [15]
- Compatibilità cross-browser e cross-device: più volte sottolineata come metrica qualitativa. Halsas (2017) ha confrontato diversi engine HTML5 non solo in termini di prestazioni, ma anche di compatibilità su Chrome, Firefox e Safari. [16]

[11] Barczak e Woźniak, «Comparative Study on Game Engines».
[12] Pattrasitidecha, *Comparison and Evaluation of 3D Mobile Game Engines*.
[13] Tufegdžić et al., «Application of WebAssembly Technology in High-Performance Web Applications».
[14] Berdak e Plechawska-Wójcik, «Performance Analysis of Unity3D Engine in the Context of Applications Run in Web Browsers».
[15] Bi et al., «Demystifying Mobile Extended Reality in Web Browsers».
[16] Halsas, *Comparison of HTML5 Game Engines Used in Game Development*.

### 2.3.2 Metodologie sperimentali dai paper analizzati

Dai paper analizzati emergono alcune metodologie ricorrenti:

- Sviluppo di prototipi equivalenti: quasi tutti gli studi sviluppano piccoli giochi o demo comparabili. Barczak e Woźniak (2019) hanno implementato lo stesso minigioco in Unity, Unreal e CryEngine per confrontare tempi di rendering e FPS . Analogamente, Halsas (2017) ha creato giochi 2D simili su più engine HTML5 per valutare fluidità e usabilità. [17] [18]
- Test su scenari controllati: i benchmark sono spesso condotti su configurazioni hardware definite e ripetibili. Berdak e Plechawska-Wójcik (2017) hanno eseguito test di Unity WebGL sia su PC desktop che su dispositivi mobili per evidenziare differenze di performance. [19]
- Uso di profiler e strumenti integrati: diversi lavori utilizzano i tool nativi degli engine (Unity Profiler, Unreal Insights) o strumenti di monitoraggio del browser. Mohd et al. (2023) evidenziano che i profiler integrati permettono di monitorare in dettaglio consumo CPU, memoria e draw calls. [20]
- Metriche quantitative e qualitative: oltre a dati numerici (FPS, memoria), alcuni studi includono valutazioni qualitative. Ad esempio, Mohd et al. (2023) hanno analizzato i punti di forza e debolezza dei principali engine moderni combinando parametri tecnici (grafica, ottimizzazione, scalabilità) con fattori pratici come la facilità d'uso, la disponibilità di documentazione e il supporto della community. [21]
- Confronto con benchmark nativi: alcuni lavori confrontano il deploy web con l'equivalente nativo per misurare la perdita prestazionale. Tufegdžić et al. (2024) hanno confrontato performance WebAssembly vs. C++ nativo in applicazioni grafiche, mostrando differenze tra il 5% e il 30%. [22]

[17] Barczak e Woźniak, «Comparative Study on Game Engines».
[18] Halsas, *Comparison of HTML5 Game Engines Used in Game Development*.
[19] Berdak e Plechawska-Wójcik, «Performance Analysis of Unity3D Engine in the Context of Applications Run in Web Browsers».
[20] Mohd et al., «Analyzing Strengths and Weaknesses of Modern Game Engines».
[21] Mohd et al., «Analyzing Strengths and Weaknesses of Modern Game Engines».
[22] Tufegdžić et al., «Application of WebAssembly Technology in High-Performance Web Applications».

# 3. Analisi degli Engine Selezionati

In questa sezione si andrà ad analizzare singolarmente ogni game engine selezionato per la sperimentazione, così da effettuare una panoramica di pregi e difetti di ognuno di essi.

## 3.1 Criteri di selezione (tecnici, di usabilità, di licenza)

Il criterio di selezione tecnica preliminare è stato indubbiamente la capacità di sviluppare un videogioco 3d, con l'obiettivo di successivamente esportarlo su web, utilizzando tecnologie di rendering come WebGL o WebGPU.

La compatibilità con WebXR rappresenta indubbiamente un aspetto di notevole interesse, in considerazione della rapida evoluzione di tale tecnologia. Tale compatibilità potrebbe risultare vantaggiosa per le aziende, in quanto permetterebbe di sviluppare applicazioni VR basate su web con maggiore facilità e celerità.

La selezione di un engine con editor visuale e, preferibilmente, piccole utility per la gestione di modelli 3D, texture e altri elementi simili si è rivelata un fattore di cruciale importanza. La presenza di documentazione, community e tutorial online è risultata un elemento altrettanto significativo. Anche la curva di apprendimento è stata un fattore determinante nella scelta, al fine di comprendere le difficoltà che si sarebbero presentate durante la transizione di un intero team di sviluppo da un motore di gioco a un altro.

I criteri di licenza, d'altro canto, sono stati concepiti principalmente per garantire che l'engine fosse utilizzabile gratuitamente, con eventuali licenze commerciali. Comunque si cerca di privilegiare gli engine open source o gratuiti nell'uso.

## 3.2 Panoramica degli engine scelti

Adesso andiamo ad analizzare singolarmente gli engine evidenziando limiti e potenzialità.

### 3.2.1 Unity 6.2

Unity è il game engine multipiattaforma di riferimento nel settore dello sviluppo di videogiochi e applicazioni interattive, noto per la sua flessibilità, facilità d'uso e per l'estesa gamma di strumenti integrati. È anche l’attuale engine utilizzato dall’azienda “Tiny Bull” per lo sviluppo delle proprie applicazioni.

Unity è un ambiente di sviluppo che consente di creare contenuti sia in modalità 2D che 3D. Questo software offre un editor visuale intuitivo, sistemi di fisica, animazione, illuminazione avanzata e una gestione delle risorse tramite un vasto store di asset. L'engine fornisce componenti per lo scripting in C#, supporta soluzioni di workflow collaborativo (Plastic SCM, Collaborate) e consente il deployment su oltre 20 piattaforme, tra cui PC, Mac, Web, console, dispositivi mobili, XR e altre.

Unity continua a evolvere, focalizzandosi su performance, stabilità e funzionalità innovative. Le versioni più recenti introducono strumenti di editing automatico, generazione di asset, supporto nativo per CoreCLR (la runtime C# più aggiornata di Microsoft), strumenti di ottimizzazione anche per grandi progetti, nuove pipeline di rendering e funzionalità avanzate per animazione, networking, LiveOps e gestione multiplayer.

L'accessibilità dell'interfaccia visuale e la documentazione dettagliata rendono Unity uno strumento adatto sia ai principianti che ai professionisti.

La presenza di un asset store, la piattaforma ufficiale che offre una vasta gamma di risorse, plugin e strumenti aggiuntivi, è un fiore all'occhiello per l'engine che agevola il riuso di risorse e permette così una veloce iterazione delle idee.

Unity inoltre permette un deploy capillare, da un solo codice è molto facile esportare poi su più piattaforme, ovviamente apportando le singole rifiniture dove necessario.

L'ecosistema è solido e la community è molto attiva, favorendo il supporto rapido e la condivisione delle migliori pratiche.

Unity si conferma nel 2025 come un sistema moderno, solido e orientato sia al mercato indie sia a quello enterprise, con continui investimenti in tecnologie che semplificano la produzione di esperienze interattive di alta qualità.

La distribuzione di giochi e applicazioni con Unity ha subito un'evoluzione fondamentale grazie all'arrivo della versione 6.2, che introduce la compatibilità con WebGPU e una serie di ottimizzazioni mirate proprio alla pubblicazione su web. Tradizionalmente, Unity si è affidato a WebGL per portare i progetti su browser, sfruttando HTML5 come ambiente di esecuzione e garantendo la compatibilità con la stragrande maggioranza dei dispositivi desktop e mobile. Ora, grazie all'introduzione di WebGPU, una tecnologia grafica che offre performance e flessibilità paragonabili a quelle delle applicazioni native, il motore rende possibile pubblicare esperienze visive di alta qualità e giochi complessi anche su web, con una fluidità che fino a poco tempo fa era impensabile. Va sottolineato che WebGPU è ancora in fase sperimentale sui browser più moderni, ma rappresenta un salto generazionale per chi mira a sfruttare il massimo delle potenzialità grafiche direttamente online.

Per garantire agli sviluppatori un percorso di pubblicazione ottimizzato, Unity 6.2 ha introdotto i Build Profiles preconfigurati per il web, potenti strumenti che semplificano la configurazione delle impostazioni di export. Questi profili sono progettati per facilitare il processo sia su desktop che su mobile, permettendo di calibrare le prestazioni e ridurre i problemi di compatibilità grazie a scelte tecniche già testate e validate. Di conseguenza, il risultato è un'esperienza utente più fluida, con impostazioni che si adattano automaticamente alla varietà di dispositivi su cui viene fruito il gioco. Un altro passo avanti riguarda l'ottimizzazione del peso e delle prestazioni delle build: Unity introduce nuove tecniche di "stripping" del codice, che permettono di eliminare moduli e componenti non utilizzati, riducendo il carico di download, i tempi di caricamento e offrendo prestazioni superiori sia su browser desktop che mobile.

L'interfaccia e la presentazione delle applicazioni Unity sul web beneficiano anche dell'introduzione dei Web Templates, ossia modelli personalizzabili pensati appositamente per l'integrazione in pagine web e siti. Questo sistema consente agli sviluppatori di adattare l'aspetto della pagina di caricamento, arricchendola con elementi di branding o funzioni aggiuntive, rendendo l'embedded di Unity veramente versatile e coerente con l'identità del prodotto. L'attenzione alle esigenze del pubblico mobile si traduce inoltre in una cura particolare nell'ottimizzazione delle build per browser su smartphone e tablet: Unity offre raccomandazioni tecniche e configurazioni dedicate, così i giochi risultano più responsivi, leggeri e stabili, mantenendo frame rate elevati anche sulle piattaforme meno potenti.

Nel caso di Unity 6.2, l'utilizzo del motore è legato all'attivazione di una licenza: ne esistono diverse tipologie, dalla versione Personal, gratuita per chi mantiene ricavi annuali sotto un certo limite, fino ai piani Pro e Enterprise pensati per professionisti e aziende. La licenza va attivata tramite processi dedicati e può essere gestita in modo manuale o automatico. È importante sapere che le condizioni variano in base al fatturato annuale generato tramite progetti Unity, e lo

sviluppo per alcune piattaforme, come console di gioco, richiede piani particolari. Unity tutela i propri diritti sui tool e sull'editor, pur lasciando piena proprietà ai creatori sui contenuti realizzati con il motore, salvo le restrizioni contrattuali che impongono la scelta della licenza adatta alla propria situazione professionale.

### 3.2.2 Godot 4.4/4.5

Godot è un motore di gioco open source che si distingue per la sua flessibilità, la modularità e la facilità d'uso, rendendolo adatto sia alla realizzazione di giochi 2D e 3D di ogni complessità. Le sue funzionalità sono progettate per fornire agli sviluppatori una piattaforma avanzata, personalizzabile e senza costi di licenza o royalty.

Al centro del sistema vi è un sistema basato su nodi e scene: ogni progetto è strutturato come un insieme gerarchico (un "albero") di nodi, organizzati in scene riutilizzabili. Ogni nodo è deputato a una funzione specifica, che può riguardare la visualizzazione di sprite, la gestione di input, suoni, fisica, animazioni o elementi UI. La struttura in esame favorisce la modularità e la manutenzione del codice: una singola scena può rappresentare un personaggio, un livello, un'arma, un elemento della UI e può essere annidata o ereditata per creare varianti o composizioni più complesse.

Godot presenta anche un editor visuale, caratterizzato da un sistema di drag & drop, che include strumenti per la creazione di contenuti bidimensionali, tridimensionali, animazioni, mappe di tile, shader e profili di performance. Si evidenzia altresì la presenza di tool integrati per il debugging, la profilazione e la gestione delle risorse.

Il sistema di scripting è flessibile e dispone di un linguaggio proprio (GDScript, simile a Python e ottimizzato per i giochi), ma è in grado di supportare anche C#, VisualScript e, grazie a GDExtension, altri linguaggi compilati come C++ o Rust.

L’engine è dotato di un motore fisico avanzato e di un sistema di rendering qualitativo in 2D e 3D. Offre inoltre un sistema di input unificato supporta la tastiera, il mouse, il gamepad, il touch.

Godot si distingue per la sua accessibilità, la libertà creativa e la rapidità con cui è possibile prototipare e iterare giochi. È uno dei pochi motori che realmente rispecchia il principio di "community driven", garantendo agli sviluppatori un controllo e una proprietà totale del proprio lavoro.

Il deploy web in Godot si configura come una soluzione efficace e diretta per la distribuzione di giochi accessibili da qualsiasi browser moderno, senza che l'utente debba installare alcunché. Nel processo di esportazione di un progetto per il web, Godot impiega WebAssembly e WebGL 2.0, tecnologie ampiamente supportate dai principali browser, al fine di garantire la portabilità e prestazioni accettabili. Il processo di esportazione è concepito per essere intuitivo: una volta configurata la piattaforma "Web" tra le opzioni di esportazione dell'editor, viene generato un pacchetto di file HTML, JavaScript e WebAssembly, che può essere ospitato su un qualsiasi server web.

A partire dalla versione 4.3 di Godot, la modalità di esportazione single-threaded è diventata la modalità predefinita, in quanto garantisce un livello di compatibilità maggiore su piattaforme diverse, inclusi macOS, iOS e servizi di hosting web come itch.io o Netlify. La modalità in questione consente di eliminare le problematiche associate ai requisiti di sicurezza e ai cors origin headers che sarebbero altrimenti necessari per il multithreading, semplificando significativamente il processo di pubblicazione. L'esperienza d'uso prevede che l'applicazione venga caricata in una pagina HTML predefinita, ma l'utente ha la possibilità di personalizzare il wrapper HTML per integrare il gioco in siti esistenti o aggiungervi funzioni supplementari. La

gestione dei file esportati richiede un'attenta analisi: il nome del file principale, generalmente index.html, deve essere coerente per garantire l'avvio corretto da web server standard. Inoltre, per i progetti che impiegano GDScript non sussistono restrizioni relative al deployment web, mentre l'uso di C# non è attualmente supportato su questa piattaforma in Godot 4.x.

La sperimentazione avverrà utilizzando anche una versione beta di Godot 4.5 che secondo un articolo di Adam Scott[23] dovrebbe implementare dei miglioramenti per quanto riguarda il deploy web perché viene migliorata la tecnologia WASM SIMD (WebAssembly Single Instruction, Multiple Data). Questa tecnologia consente alla CPU di eseguire operazioni parallele su più dati contemporaneamente, incrementando la velocità dei calcoli effettuati dal gioco, in particolare nelle situazioni più complesse e onerose dal punto di vista computazionale, come ad esempio le simulazioni fisiche caotiche. I test di benchmark condotti su macchine potenti hanno evidenziato incrementi significativi, fino a 10-15 volte in alcune condizioni estreme di carico, anche se in situazioni normali i miglioramenti sono più modesti (circa il doppio). Questo implica che, in condizioni di elevata intensità di calcolo o di gestione di molteplici entità, Godot 4.5 con WASM SIMD è in grado di garantire un frame rate più fluido e stabile, evitando i drastici cali di prestazioni che si otterrebbero in assenza di tale tecnologia.

Seppure non offrendo in termini di perfomance ottime prestazioni, la soluzione di deploy web offre indubbi vantaggi in termini di accessibilità immediata e possibilità di aggiornare rapidamente i propri progetti, sia tramite hosting autonomo sia tramite piattaforme come Netlify, Vercel o itch.io. Su dispositivi mobili, la compatibilità è garantita, sebbene con alcune limitazioni rispetto alle build native. Pertanto, è sempre consigliabile ottimizzare le risorse e il carico computazionale per evitare rallentamenti. In sintesi, il deploy web di Godot si distingue per immediatezza, praticità e coerenza rispetto alla filosofia open e accessibile dell'engine, offrendo uno strumento solido sia per il testing rapido che per la distribuzione pubblica di giochi e applicazioni interattive.

Godot rappresenta un esempio emblematico di software libero, distribuito con licenza MIT. Questa licenza garantisce piena libertà non solo nell’uso del motore, ma anche nella sua modifica, redistribuzione e integrabilità in altri progetti, siano essi open source o commerciali. L’unico vincolo impone di citare la licenza e la provenienza del software laddove venga utilizzato o riutilizzato il motore. Da sottolineare che la licenza MIT non si applica automaticamente ai progetti creati con Godot, che restano di esclusiva proprietà dell’autore, il quale può scegliere liberamente qualsiasi modello di licensing per il suo gioco o applicazione, senza vincoli né royalty dovute agli sviluppatori originali.

### 3.2.3 PlayCanvas

PlayCanvas si presenta come un game engine all'avanguardia e altamente specializzato per la creazione di esperienze interattive 3d direttamente all'interno del browser, distinguendosi per la sua propensione web-native in ogni componente della sua struttura e del suo flusso di lavoro, questo engine appunto è interoperabile solamente tramite browser.

Sin dalla sua fase di ideazione, PlayCanvas è stato concepito non solo come motore open source, scritto in JavaScript e perfettamente integrato con le tecnologie web moderne come WebGL e HTML5, ma anche come piattaforma collaborativa nella quale l'editor può essere utilizzato via browser, senza necessità di installazioni locali. Questa caratteristica consente una produttività agile e in tempo reale: più sviluppatori possono lavorare simultaneamente sullo stesso progetto, vedere immediatamente i cambiamenti e condividere facilmente progetti e risorse. L'approccio

[23] Engine, «Upcoming (Serious) Web Performance Boost».

JavaScript/TypeScript garantisce un'elevata accessibilità sia agli sviluppatori esperti che ai neofiti del mondo della programmazione web, facilitando la curva di apprendimento e la creazione di estensioni o personalizzazioni.

In termini di potenzialità, PlayCanvas integra una fisica 3D, supportata dal motore ammo.js e dall'implementazione di funzionalità quali rigid body, collisioni, suoni 3D posizionali e animazioni complesse, insieme a un sistema di rendering WebGL che consente effetti visivi di alta qualità e una gestione avanzata di materiali e luci. L'engine supporta asset standard quali glTF, gestisce texture ed effetti post-processing e consente la pubblicazione di applicazioni che vanno dal gaming all'advertising, dalla realtà virtuale all'architettura interattiva, fino a musei digitali e configuratori.

In confronto ad altri engine, PlayCanvas si distingue principalmente per la sua leggerezza, per i tempi di caricamento rapidi e per la capacità di scalare efficacemente anche su dispositivi mobili, grazie a diversi strumenti di ottimizzazione come il caricamento differito degli asset e la minificazione del codice.

Nel contesto della pubblicazione e del deployment web, PlayCanvas emerge come una soluzione particolarmente immediata, flessibile e performante tra i motori per browser. Una volta completato il progetto nell'editor online, è possibile pubblicarlo con un unico clic sulla piattaforma PlayCanvas, ottenendo immediatamente un URL pubblico e pronto all'uso. Questo processo riduce la complessità della condivisione e del testing in tempo reale con clienti, colleghi o utenti finali, eliminando la necessità di configurazioni server o distribuzioni di file. Inoltre, PlayCanvas consente l'esportazione del progetto come archivio .zip contenente tutti i file necessari per l'hosting su server proprietari, CDN o piattaforme di cloud hosting come Netlify e Vercel, inclusi file HTML, JavaScript, asset e impostazioni. Il motore fornisce documentazione dettagliata su come gestire la distribuzione self-hosted, ottimizzando cache, compressione e performance dei contenuti statici, e persino la possibilità di integrare i giochi tramite semplice iframe nei propri siti web. Tutte queste opzioni conferiscono a PlayCanvas un'elevata versatilità per chi desidera una distribuzione massiccia su web, sia per progetti dimostrativi, giochi ad alto traffico o soluzioni commerciali.

La capacità di PlayCanvas di supportare un deploy web senza soluzione di continuità è strettamente correlata alla sua architettura web-first: le applicazioni funzionano nativamente nel browser senza plugin o runtime esterni, sfruttano appieno le evoluzioni di JavaScript per performance e compatibilità, e ricevono continui aggiornamenti dalla community e dal team di sviluppo per inglobare le ultime novità in ambito rendering (supporto a WebGPU per esempio), networking e UX sul web.

PlayCanvas possiede uno spirito open source: il cuore del motore è libero e disponibile sotto licenza MIT, permettendo totale flessibilità, sia per progetti commerciali che privati. Il motore, scritto in JavaScript, può essere scaricato, modificato e integrato senza vincoli, abilitando anche la distribuzione di versioni personalizzate. Sono disponibili servizi aggiuntivi e strumenti commerciali soprattutto in versione SaS, ma l’engine di base resta utilizzabile senza costi (con alcuni limiti sul salvataggio dei progetti), favorendo la sperimentazione e la crescita della community. Questa apertura ha permesso a PlayCanvas di essere adottato non solo da sviluppatori indie, ma anche da grandi aziende per applicazioni professionali, giochi e visualizzazioni 3D su web

### 3.2.4 Rogue Engine

Rogue Engine si configura come una soluzione innovativa per la creazione di videogiochi e applicazioni interattive 3d, con un approccio orientato specificatamente all'ambiente web e alla

massima accessibilità tra le diverse piattaforme. Basato sulle potenzialità di Three.js, il motore è concepito per sviluppatori e web designer che desiderano dare vita alle proprie idee sfruttando le tecnologie web più moderne, senza rinunciare a un editor visivo utilizzabile tramite desktop, simile a quanto offrono editor come Unity.

L'architettura di Rogue Engine promuove l'impiego di TypeScript e JavaScript, integrando in modo ottimale il flusso di lavoro con Node.js e permettendo la gestione di pacchetti esterni tramite NPM (Node Package Manager). L'interfaccia editor è intuitiva ma molto acerba, dispone di una modalità visuale, che consente di costruire scene, gestire asset 3D, illuminazione, materiali, animazioni e componenti.

Un aspetto di notevole rilevanza è rappresentato dalla modularità intrinseca del sistema, che consente la creazione di componenti personalizzati con interfacce semplici, facilmente riutilizzabili o condivisi tra progetti, promuovendo così la collaborazione reciproca e la costruzione di un proprio ecosistema di strumenti e script.

Sono disponibili integrazioni per funzionalità complesse quali la modalità multiplayer (tramite Croquet), l'animazione, le simulazioni fisiche aggiuntive e la gestione di asset, attraverso un package manager che supporta sia plugin gratuiti che premium. L'approccio "fair license" di Rogue Engine consente l'utilizzo gratuito per la maggior parte dei creatori, applicando tariffe solo per coloro che superano una soglia di guadagno. Questo modello mantiene la proprietà intellettuale dei progetti esclusivamente nelle mani dell'autore, eliminando numerosi vincoli commerciali.

Per quanto concerne il deployment su web, Rogue Engine è stato concepito e ottimizzato proprio per soddisfare tale requisito. Una volta sviluppata la scena o il gioco, il processo di build restituisce un pacchetto leggero ed estremamente rapido da avviare su qualsiasi server statico o piattaforme di hosting specializzate come Itch.io, Poki e altre. In alternativa, è disponibile il servizio Rogue Play, che consente la pubblicazione e l'hosting istantanei con pochi clic, favorendo la distribuzione immediata e l'accesso universale tramite URL pubblico. Il motore genera asset e codice ottimizzato per browser moderni, sfruttando appieno WebGL e tutte le risorse di accelerazione grafica disponibili, rendendo i prodotti fruibili non solo su desktop ma anche su dispositivi mobili e VR standalone, senza compromettere le prestazioni.

L'approccio "web-native" di Rogue Engine consente di gestire il deployment come una conseguenza naturale del workflow, rendendo superfluo l'impiego di conversioni, launcher o plugin esterni. Ogni progetto è concepito per funzionare direttamente all'interno del browser, eliminando la necessità di ulteriori passaggi. Questo rende Rogue Engine uno dei pochi sistemi che offre un'esperienza utente realmente fluida per la diffusione delle proprie creazioni tridimensionali sulla rete, garantendo tempi di caricamento rapidi, compatibilità assicurata e la possibilità di integrare facilmente le applicazioni in siti web, servizi cloud o store digitali.

Rogue Engine, infine, adotta una filosofia “fair license”, pensata per combinare apertura e sostenibilità. L’utilizzo personale e commerciale è consentito senza costi fino al raggiungimento di una soglia di ricavi, fissata a 80.000 dollari in 12 mesi per la licenza base. Superata questa soglia, sono disponibili piani a pagamento pensati per professionisti, aziende e team di ogni dimensione, che aggiungono funzionalità premium e servizi di supporto. Fondamentale è il diritto garantito agli autori di mantenere la piena proprietà sui contenuti creati, vincolando solo la ridistribuzione o la commercializzazione diretta del motore stesso, non quella dei prodotti realizzati; in questo modo si tutela lo sviluppo indipendente abbattendo sia vincoli di royalty che restrizioni sulla libertà creativa.

## 3.3 Tabella comparativa delle feature

Di seguito una tabella comparativa che riassume tutte le specifiche elencate in precedenza:

| **Criterio** | **Unity 6.2** | **Godot** | **PlayCanvas** | **Rogue Engine** |
|---|---|---|---|---|
| Licenza | Commerciale con vari piani (Personal, Pro, Enterprise), attivazione necessaria, basata su fatturati | Licenza MIT open source, libero di usare, modificare e distribuire | Licenza MIT open source, libero di usare e modificare. Possibilità di utilizzo SaS | Fair License: gratis fino a 80k$ ricavi/anno, poi licenza commerciale |
| Linguaggi Principali | C#, visual scripting | GDScript, C#, C++, VisualScript | JavaScript, TypeScript | JavaScript, TypeScript |
| Piattaforme Deploy | Desktop, Mobile, Web, Console | Desktop, Mobile, Web, Console | Web | Web |
| Open Source | No | Sì | Sì | Parziale |
| Fisica | Motore fisico avanzato (PhysX), supporto soft body, ragdoll, cloth | Bullet & Godot Physics, collisioni ottimizzate, soft body limitato | Supportata tramite integrazione con ammo.js | Supporto fisica solo tramite librerie esterne |
| Animazioni | Timeline editor, animations 2D/3D complesse, blending, rigging avanzato | Editor animazioni integrato, rigging base, blending, animazioni 2D/3D con keyframes | Animazioni scheletriche base, blending limitato | Supporta le animazioni, blending limitato |
| Distribuzione Web | Build ottimizzate WebGL/WebGPU, strumenti per personalizzazione e ottimizzazione mobile | Export WebAssembly semplice, WASM SIMD per migliori prestazioni | Pubblicazione diretta online o export per hosting | Web-native, build leggere ottimizzate per hosting diretto e instantaneo |

# 4. Metodologia Sperimentale

In questa sezione si andrà a esplicare quale sarà il prototipo creato per lo studio e le metodologie di raccolta dei dati.

## 4.1 Struttura generale del benchmark

La progettazione di un protocollo di benchmark rappresenta un elemento importante per questa sperimentazione, in quanto consente di confrontare in modo equo e riproducibile i diversi motori di gioco selezionati. La struttura sperimentale è stata definita facendo riferimento sia ai contributi presenti in letteratura scientifica sia alle specifiche esigenze del progetto, in collaborazione con Tiny Bull Studios. L'obiettivo del progetto era la creazione di un ambiente di test standardizzato, in grado di isolare l'impatto del motore di gioco dalle altre variabili e di ottenere dati quantitativi e qualitativi confrontabili.

Nella bibliografia scientifica, sono presenti diversi approcci al benchmarking degli engine. A titolo esemplificativo, Barczak e Woźniak (2019) hanno implementato lo stesso minigioco in Unity, Unreal e CryEngine[24], confrontando il comportamento dei tre motori in termini di frequenza dei fotogrammi e utilizzo delle risorse. In un diverso studio, Berdak e Plechawska-Wójcik (2017) hanno focalizzato la loro attenzione sulle differenze prestazionali tra le build native e le build WebGL di Unity, includendo misure di tempo di caricamento, frequenze di elaborazione e memoria occupata[25]. Studi più recenti hanno messo in evidenza la necessità di integrare metriche relative all'esperienza dell'utente, quali il First Load Time (FLT) e la compatibilità cross-browser e cross-device[26].

In conformità con le suddette premesse, la struttura del benchmark sviluppata in questo lavoro si articola nei seguenti elementi:

1. Si è proceduto alla definizione di un caso di studio comune. È stato progettato un gioco tridimensionale di tipo endless runner, denominato "Knight Runner", che rappresenta una tipologia diffusa nel settore del web gaming e, al tempo stesso, particolarmente adatta a stressare i motori di gioco sia dal punto di vista grafico sia da quello logico. L'impiego di un caso di studio comune consente di ridurre al minimo le differenze legate al design e di concentrare l'analisi sull'engine.
2. Si è effettuata una implementazione su ogni engine cercando di mantenere il più possibile un'implementazione simile del gioco, rendendolo giocabile e funzionante apportando opportuni aggiustamenti successivamente documentati.
3. Si è effettuata una raccolta dei dati automatica mentre il gioco veniva eseguito sui vari dispositivi di test raccogliendo dati come: frame rate campionato ogni 3 secondi e successivamente analizzato per ottenere una media, minimo e massimo; tempo di caricamento e compatibilità cross-browser e cross-device.

Per quanto concerne gli scenari di test e ripetibilità, si informa che ogni motore è stato sottoposto a test in condizioni identiche su un insieme predefinito di browser e dispositivi, al fine di garantire la comparabilità dei risultati. Le sessioni di gioco sono state programmate con una durata prestabilita e uguale per tutti gli esperimenti.

La struttura del benchmark adottata rappresenta pertanto un compromesso tra rigore scientifico e applicabilità pratica. Da un lato, tale approccio si fonda su criteri metodologici consolidati nella

[24] Barczak e Woźniak, «Comparative Study on Game Engines».
[25] Berdak e Plechawska-Wójcik, «Performance Analysis of Unity3D Engine in the Context of Applications Run in Web Browsers».
[26] Bi et al., «Demystifying Mobile Extended Reality in Web Browsers».

letteratura scientifica, garantendo la validità delle misure; dall'altro, considera le esigenze industriali, assicurando che i risultati possano essere immediatamente applicati per orientare decisioni di sviluppo concrete.

## 4.2 Gioco 3D Endless Runner: "Knight Runner"

Al fine di valutare in maniera comparativa le prestazioni degli engine selezionati, è stato scelto di sviluppare un prototipo tridimensionale di tipo endless runner, denominato "Knight Runner". La scelta di tale tipologia di gioco è motivata dalla sua ampiamente diffusa presenza nel panorama videoludico mobile e web. Tale genere si distingue per una struttura semplice, ma al tempo stesso in grado di generare un carico costante e ripetibile sul motore grafico e logico.

Dal punto di vista estetico, il prototipo adotta uno stile grafico low-poly di tipo cartoon, realizzato mediante l'impiego di asset disponibili in librerie open access. La costruzione dell'ambiente di gioco, in particolare il “blocking” e la disposizione modulare delle piattaforme, è stata realizzata mediante l'utilizzo del software Blender, per poi essere esportata e integrata nei diversi engine. L'approccio metodologico selezionato garantisce una base coerente per l'analisi di tutti i motori esaminati, assicurando la comparabilità dei risultati.

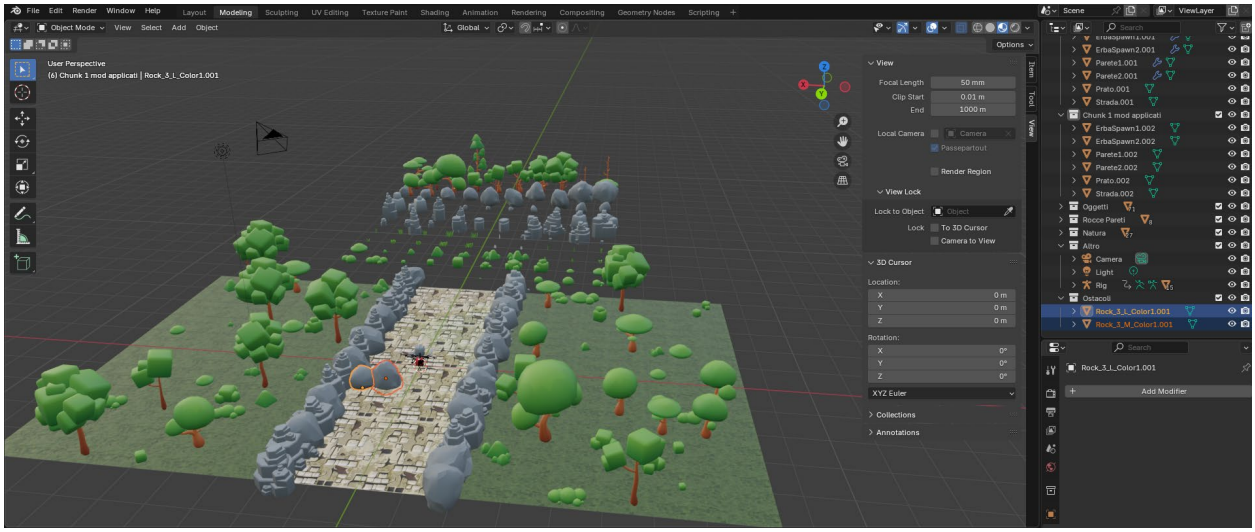


*Figura 6 - Blocking su Blender*

Il gameplay prevede l'impiego di un cavaliere come personaggio principale, dotato di un set ridotto ma funzionale di animazioni (idle, corsa e salto). Una scelta metodologica rilevante riguarda l'implementazione del movimento: il personaggio non si sposta nello spazio in avanti ma solo a destra e a sinistra nella scena, è l'ambiente circostante a scorrere davanti ad esso.

Gli ostacoli presenti nel gioco sono rappresentati da delle rocce, disposte sul percorso in maniera e in numero casule alla generazione della piattaforma; se queste colpiscono il personaggio scaturiscono un game over. Questa logica minimale consente di testare i sistemi di fisica e di collisione in modo basilare, senza introdurre variabili legate a comportamenti avanzati degli NPC (Non Player Character) o ad ambienti complessi.

Il sistema di punteggio è basato sulla distanza percorsa calcolata come numero di piattaforme superate, introduce un elemento di progressione sufficiente a rendere il prototipo "giocabile", pur mantenendo il focus sulla ripetibilità e sulla standardizzazione dei test.

Dal punto di vista tecnico, sono stati impiegati modelli e texture low-poly, senza l'integrazione di effetti particellari o sistemi di illuminazione avanzata.

L'applicazione della fisica nel gioco è stata effettuata in modo minimale, limitandosi al calcolo delle collisioni tra il cavaliere e gli ostacoli, utilizzando i sistemi nativamente offerti dai motori. Questo approccio ha consentito di valutare il comportamento delle implementazioni di fisica di ciascun motore senza introdurre differenze significative nelle condizioni di test. In tale prospettiva, si è deliberatamente optato per un approccio neutrale e comparabile, piuttosto che per la ricerca del massimo dettaglio grafico o della fedeltà simulativa.

Al fine di garantire l'omogeneità del confronto, è stato deciso di utilizzare un setup base e un export "vanilla" delle build web, evitando l'introduzione di ottimizzazioni specifiche per i singoli engine. Tale scelta, se da un lato non consente di valorizzare appieno le potenzialità di ciascuna piattaforma, dall'altro permette di evidenziare i limiti strutturali e le prestazioni "out-of-the-box", condizione particolarmente rilevante per gli sviluppatori che devono prendere decisioni rapide in fase di selezione della tecnologia.

Il prototipo "Knight Runner" è stato quindi implementato in tutti gli engine esaminati nello studio: Unity, Godot, PlayCanvas e Rogue Engine. Nei casi in cui è stato possibile (Unity e PlayCanvas), sono state generate anche build WebGL e WebGPU alternative, al fine di analizzare il comportamento dei diversi motori di rendering. Tutti i prototipi sono stati pubblicati sulla piattaforma itch.io, che è stata selezionata come ambiente di deploy per la sua diffusione e per la possibilità di condurre test diretti su un'ampia gamma di dispositivi e browser. La disponibilità online delle build ha inoltre favorito la riproducibilità dell'esperimento, consentendo di estenderlo a diversi contesti hardware e garantendo una raccolta dati più ampia e diversificata.

## 4.3 Parametri controllati nell’ambiente di test

Al fine di garantire la comparabilità dei risultati tra i diversi motori di gioco, è stato definito un insieme di parametri controllati che hanno regolato le condizioni sperimentali. Il controllo delle variabili rappresenta un aspetto cruciale per un'analisi scientifica delle prestazioni, in quanto consente di isolare l'impatto dell'engine dalle differenze dovute alla configurazione grafica, alla complessità logica o al dispositivo utilizzato. In conformità con gli studi precedenti, l'obiettivo principale è stato quello di sviluppare scenari di test omogenei e riproducibili, pur mantenendo una coerenza con le dinamiche di un prototipo giocabile.

### 4.3.1 Variabili grafiche (risoluzione, qualità, complessità scena)

Per quanto concerne la configurazione grafica, la risoluzione è stata stabilita in modo da corrispondere a quella nativa del dispositivo in modalità a schermo intero, al fine di riprodurre l'esperienza effettiva di un utente finale. Non sono stati definiti profili di qualità multipli (ad esempio "basso, medio, alto"), ma è stato adottato un unico livello grafico per tutti i motori, al fine di minimizzare le differenze introdotte da impostazioni personalizzate.

La complessità della scena è stata mantenuta costante su tutti i motori di rendering: il numero di oggetti visibili sullo schermo è stato calibrato in modo da risultare comparabile, evitando variazioni che avrebbero potuto alterare la percezione del carico computazionale.
L'illuminazione è stata implementata in forma basilare, con una singola luce direzionale, senza effetti avanzati di ombreggiatura o riflessione, al fine di focalizzare i test sulle performance del motore di rendering anziché sulla gestione di sistemi grafici complessi.

### 4.3.2 Variabili di carico logico (NPC, oggetti, fisica)

Dal punto di vista logico, la complessità è stata mantenuta minima e standardizzata. Gli ostacoli, costituiti da rocce, vengono generati proceduralmente con quantità variabile, ma seguendo la stessa logica in tutti gli engine, al fine di garantire condizioni di carico equivalenti. Non sono stati introdotti NPC aggiuntivi né altri elementi dinamici oltre al personaggio principale e agli ostacoli, in quanto l'obiettivo principale era la concentrazione sulla fisica e sul rendering di base.

Le collisioni sono state implementate mediante sistemi basilari (box collider e rigidbody), sfruttando le librerie native di ciascun motore. Inoltre, la velocità del gameplay è stata resa variabile in funzione del punteggio: all'aumentare delle piattaforme superate, il punteggio aumenta e quindi il ritmo di esecuzione accelerava progressivamente, introducendo un incremento del carico computazionale che ha permesso di osservare come ciascun motore di gioco gestisse condizioni di stress crescente.

## 4.4 Metriche di misurazione

L'analisi comparativa dei motori di gioco è stata condotta sulla base di un insieme di metriche definite in accordo con l'azienda, in base alle loro esigenze. In particolare, le misure si sono concentrate su tre aspetti ritenuti centrali per valutare l'efficacia del deployment di videogiochi in ambiente web: le performance grafiche, il tempo di caricamento iniziale e la compatibilità cross-browser e cross-device.

### 4.4.1 Performance (FPS e frame time)

La prima categoria di metriche concerne le prestazioni di rendering, misurate mediante il frame rate e il frame time. Per ciascun esperimento, è stata effettuata una raccolta continua dei dati, con registrazioni inviate a un server privato ogni tre secondi tramite uno script integrato nel gioco. I dati sono stati organizzati in un foglio elettronico e successivamente analizzati per calcolare il valore medio, il valore massimo, il valore minimo e la deviazione standard.

L'impiego della deviazione standard, oltre al valore medio, ha consentito di misurare la stabilità del frame rate e di identificare eventuali oscillazioni o picchi critici, in conformità con le raccomandazioni di Berdak e Plechawska-Wójcik (2017), che avevano sottolineato come le medie possano occultare cali improvvisi di performance. Inoltre, dal frame rate è stato calcolato il frame time medio (in ms), che risulta essere un indicatore più significativo della fluidità percepita dall'utente (Tufegdžić et al., 2024).

### 4.4.2 Tempo di caricamento (First Load Time)

Un secondo parametro di notevole rilevanza è stato il First Load Time (FLT), ovvero l'intervallo di tempo intercorrente tra l'avvio dell'applicazione e la visualizzazione del primo frame giocabile. Il parametro in questione è stato rilevato manualmente, mediante misurazione cronometrica, a partire dal momento in cui l'utente avviava il prototipo sulla piattaforma itch.io fino alla comparsa del primo frame. Non è stata effettuata una distinzione tra tempo di download delle risorse e tempo di inizializzazione interna del motore, ma piuttosto è stato considerato il tempo totale necessario affinché il gioco diventasse effettivamente utilizzabile, questa metrica risulta di vitale importanza per l'azienda perché a volte si sono ritrovati a cercare di accorciare sempre più questo grande golfo per l'utente.

Le misurazioni sono state ripetute su ciascun dispositivo e browser coinvolto nell'esperimento, al fine di considerare le differenze derivanti dalle specifiche hardware e dall'ottimizzazione del motore rispetto all'ambiente di esecuzione.

### 4.4.3 Compatibilità cross-browser e cross-device

La terza dimensione analizzata riguarda la compatibilità dei prototipi sui diversi ambienti di esecuzione. La valutazione è stata condotta su una vasta gamma di dispositivi, inclusi portatili Windows di bassa/media gamma, PC desktop di fascia alta, tablet Android, smartphone Android di fascia bassa e alta, iPhone, iPad e un Mac con processore M1. Per quanto concerne i browser, sono stati esaminati Chrome e Firefox su Windows, i browser nativi dei dispositivi Android e iOS, nonché Safari su macOS.

La compatibilità è stata valutata in forma binaria (funziona/non funziona), registrando la capacità del gioco di avviarsi e di essere giocato sul dispositivo considerato. Eventuali problematiche di natura specifica (ad esempio, funzionalità parziali o errori grafici) verranno discusse nel capitolo dei risultati, dove sarà possibile fornire una valutazione qualitativa più approfondita.

## 4.5 Strumenti e procedure di raccolta dati

Le prove sono state condotte su un set eterogeneo di browser e dispositivi che adesso si specificano:

- Pc fisso di fascia alta con processore i5 di 13esima generazione, 16 giga di ram, scheda grafica nvdia 3060 ti;
- Pc portatile fascia medio/bassa con processore i5 di 11esima generazione, 8 giga di ram, scheda grafica integrata;
- Pc portatile Mac con processore M1;
- Tablet Android Samsung S8+;
- Tablet iPad modello 2019;
- Smartphone Android fascia alta Pixel 8;
- Smartphone Android fascia bassa Oppo A74;
- Smartphone iOs iPhone 16 pro.

Le sessioni di test hanno avuto una durata standard di 3 minuti, durante i quali il gioco veniva eseguito senza interruzioni. La durata dell'esperimento è stata selezionata per garantire la raccolta di un numero adeguato di campioni, mantenendo però la coerenza tra tutte le condizioni sperimentali. Ogni esperimento è stato replicato una sola volta per ciascuna configurazione, al fine di preservare la scalabilità della campagna di test e consentire la valutazione su un vasto numero di combinazioni tra motore di elaborazione, dispositivo e browser.

La raccolta dei dati sperimentali è stata organizzata mediante un sistema strutturato, progettato per garantire la continuità delle misurazioni e la loro successiva analisi in forma aggregata.

In ciascun motore di gioco è stato implementato uno script ad hoc, sviluppato nel linguaggio nativo dell'engine (C# per Unity, GDScript per Godot, JavaScript per PlayCanvas e Rogue Engine). Lo script in questione era programmato per monitorare i parametri prestazionali rilevanti, con un'attenzione particolare rivolta al frame rate istantaneo, e per inviare tali dati a intervalli regolari sotto forma di payload JSON a un server privato n8n predisposto per la raccolta. Il server fungeva da nodo intermedio, scrivendo i dati ricevuti in un foglio elettronico strutturato, che costituiva il repository centrale delle misurazioni.

Le misurazioni relative alle prestazioni (FPS e frame time) sono state condotte mediante campionamento continuo, con l'invio di record ogni tre secondi durante l'intera durata della sessione di test. La suddetta modalità ha consentito di ottenere serie temporali sufficientemente granulari per il calcolo di statistiche significative, quali la media, il valore minimo e massimo e la deviazione standard, e per l'evidenziazione di eventuali fluttuazioni. Successivamente, i dati quantitativi sono stati elaborati mediante funzioni di analisi direttamente disponibili nel foglio elettronico, senza la necessità di strumenti di analisi esterni.

Le misurazioni del First Load Time (FLT) e della compatibilità sono state invece raccolte manualmente. Nel primo caso, il tempo di caricamento è stato misurato mediante l'impiego di un cronometro, calcolando l'intervallo di tempo intercorrente tra l'avvio dell'applicazione sul portale itch.io e la visualizzazione del primo frame giocabile. Nel secondo caso, la compatibilità è stata valutata in termini binari (funziona/non funziona), annotando in quale combinazione dispositivo-browser il prototipo risultava effettivamente avviabile. Entrambi i set di dati sono stati inseriti nello stesso foglio di raccolta delle misurazioni, al fine di costituire un quadro integrato delle performance.

Per quanto concerne il setup hardware, i dispositivi di test sono stati impiegati nelle loro configurazioni native, senza monitoraggio aggiuntivo dei consumi hardware (CPU, GPU, RAM). Tuttavia, sono state meticolosamente annotate le specifiche tecniche principali, quali processore, memoria, scheda grafica e sistema operativo, al fine di contestualizzare i risultati e garantire la riproducibilità dei test in scenari simili.

Le sessioni di test sono state condotte sotto forma di partite di gameplay attivo, della durata di circa tre minuti ciascuna. L'interazione dell'utente, come nel caso del salto del cavaliere per evitare ostacoli, è stata presente per l'intera sessione di prova, al fine di garantire condizioni realistiche d'uso e di assicurare che le misurazioni riflettessero il comportamento del motore in un contesto di gioco effettivo, anziché in una dimostrazione priva di input reale.

# 5. Implementazione

In questa sezione andremo ad analizzare le singole implementazioni effettuate, parlando anche delle piccole ottimizzazioni necessarie per far funzionare il gioco sui diversi engine.

## 5.1 Sviluppo del prototipo nei diversi engine

### 5.1.1 Unity

La prima implementazione del prototipo Knight Runner è stata realizzata utilizzando Unity 6.2, scelto come punto di partenza in virtù della sua diffusione nel settore e della solidità delle sue pipeline di sviluppo. I modelli 3d, precedentemente organizzati in Blender per garantire coerenza tra le diverse versioni del gioco, sono stati esportati nel formato FBX, che rappresenta lo standard di maggiore compatibilità con Unity. Questa scelta ha permesso di mantenere un flusso di lavoro lineare e di minimizzare i problemi di importazione, in particolare per quanto riguarda la corretta gestione delle animazioni e delle gerarchie di oggetti.

Di seguito la struttura della scena che adesso andiamo a esplicare:

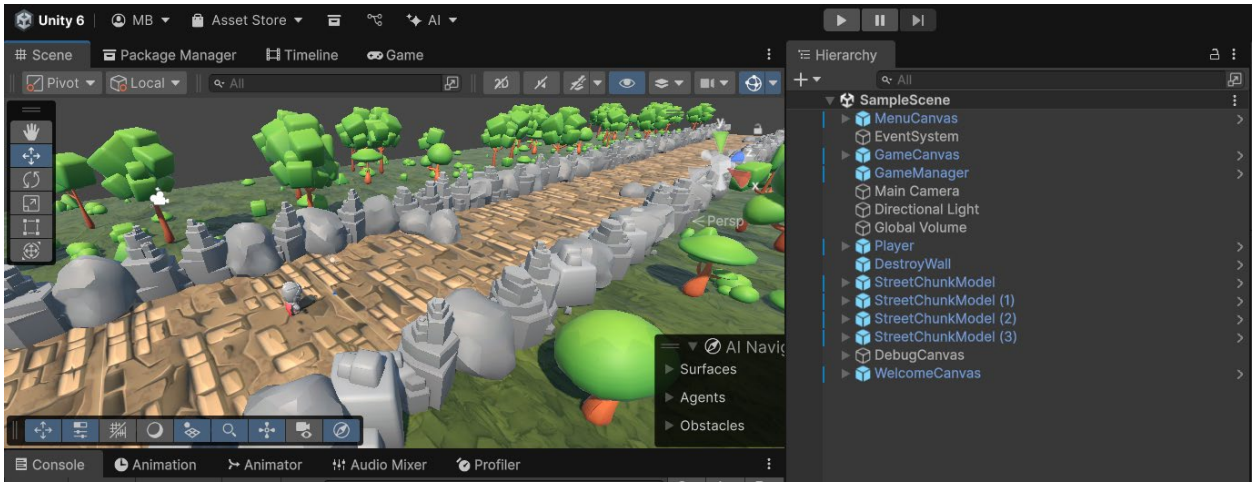



*Figura 7 - Scena principale del gioco su Unity*

La scena di gioco è stata strutturata attraverso una serie di oggetti principali, ciascuno con ruoli specifici. In primo luogo, sono stati realizzati diversi Canvas per la gestione delle interfacce utente:

- MenuCanvas costituisce l'interfaccia iniziale del gioco. Esso contiene il logo, un pulsante per avviare la partita e un'etichetta dedicata alla visualizzazione del punteggio massimo (high score), mantenendo quindi traccia della migliore performance del giocatore;


*Figura 8 - Menu principale del gioco su Unity*

- GameCanvas rappresenta invece l'interfaccia utente durante il gameplay. In questo caso l'utente visualizza il punteggio corrente, aggiornato in tempo reale dal GameManager;


*Figura 9 - Interfaccia in game su Unity*

- WelcomeCanvas appare al primo avvio e presenta all'utente un messaggio di benvenuto. Tramite l'interazione con questa UI e la pressione del pulsante "Accept" si attiva lo script di raccolta dati, che avvia la trasmissione dei parametri prestazionali al server remoto. In questo modo, l'utente viene esplicitamente reso partecipe del processo di sperimentazione.

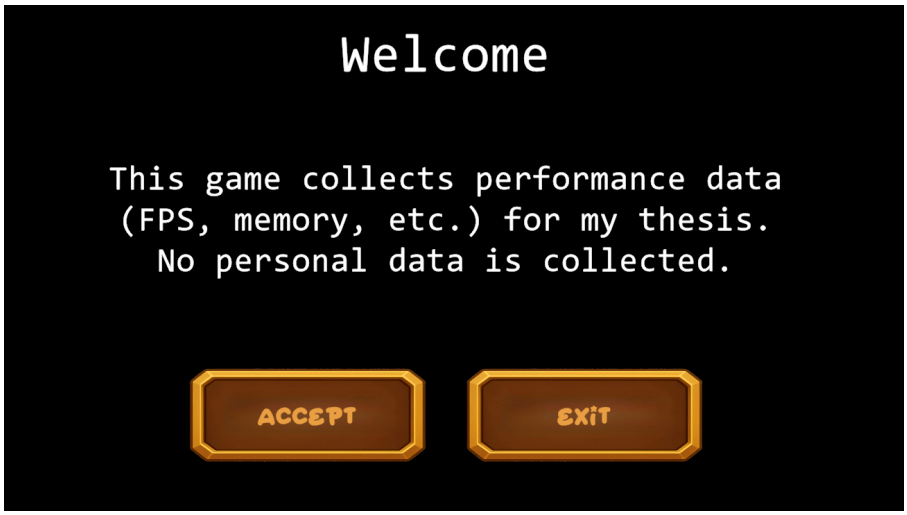



*Figura 10 - Welcome Canvas su Unity*

- DebugCanvas è l'elemento deputato alla visualizzazione e alla gestione delle informazioni sulle prestazioni utili per gli esperimenti. Tramite un apposito script, i dati di performance vengono aggiornati in UI ogni 0,25 secondi, consentendo all'utente di monitorare l'andamento del gioco. Inoltre, questo oggetto integra un ulteriore script, denominato Stats Logger, incaricato di inviare i dati prestazionali al server ogni 3 secondi. L'invio avviene tramite una richiesta HTTP (UnityWebRequest) che simula il comportamento di un form web, adottando quindi una modalità coerente con un normale flusso client–server su browser.


*Figura 11 - Debug Canvas su Unity*

Un aspetto interessante di questa implementazione riguarda il metodo di misurazione degli FPS e del frame time. Tali valori sono stati calcolati a partire dal delta time non scalato di Unity, secondo la logica riportata di seguito:

```
        float dt = Time.unscaledDeltaTime;
        float frameTime = dt * 1000f;
        int fps = Mathf.RoundToInt(1f / Mathf.Max(dt, 0.0001f));
```

In questo modo, il frame time è espresso in millisecondi, mentre gli FPS vengono derivati dal reciproco del tempo di aggiornamento, garantendo una stima affidabile anche in presenza di variazioni improvvise del carico di calcolo.

La logica di gioco è stata centralizzata all'interno del GameManager, oggetto che coordina sia il gameplay sia le interfacce utente. All'avvio di una nuova partita, il GameManager si occupa di "pulire" la scena eliminando le piattaforme residue e gli ostacoli, ricreando quindi un set iniziale privo di ostacoli. Successivamente emette un evento di inizio partita e avvia la gestione della velocità delle piattaforme, incrementandola progressivamente man mano che il punteggio cresce. Lo stesso GameManager è inoltre responsabile della gestione del punteggio e dell'aggiornamento dell'UI, nonché della gestione del game over, che comporta l'emissione di un evento di fine gioco, il ritorno al menu principale e l'aggiornamento dell'high score.

Il prefab Player contiene il modello del cavaliere protagonista. Le animazioni sono state implementate tramite l'Animator Controller di Unity, che consente di definire un albero di stati animati controllati da variabili. Questo approccio ha reso possibile gestire in modo strutturato la sequenza delle animazioni (idle, corsa e salto). Oltre all'animator, il prefab include script dedicati al movimento e alla logica del gameplay, tra cui Trigger Street Generation, responsabile della creazione dinamica del percorso: ogni volta che il cavaliere attraversa un trigger con tag "Trigger", viene generato un nuovo segmento di strada. Lo stesso script gestisce anche la collisione con oggetti contrassegnati dal tag "Obstacle", evento che determina il game over.

Il prefab del Player include quindi un box collider e un rigidbody impostato come "kinematic", così da consentire la gestione delle collisioni senza applicare simulazioni fisiche avanzate.

Il prefab StreetChunkModel rappresenta un segmento modulare del percorso, anch'esso ottenuto da un modello FBX esportato da Blender. Ogni pezzo di strada è dotato di rigidbody e di un oggetto figlio con box collider e tag "Trigger". Questo prefab include uno script Move, che gestisce il movimento della piattaforma in base allo stato del gioco, per far avviare il movimento riceve dal GameManager gli eventi di variazione della velocità e di inizio/fine gioco.

Inoltre questo prefab possiede uno script per la generazione casuale degli ostacoli. Ad ogni istanza, viene infatti definito un set di rocce posizionate in modo variabile, entro un numero massimo prestabilito, così da garantire varietà al gameplay pur mantenendo controllata la complessità della scena.

Per evitare l'accumulo di oggetti nella scena, è stato infine introdotto l'oggetto DestroyWall, posizionato dietro al giocatore dopo qualche metro. Si tratta di un trigger con tag "Destroy", quando una piattaforma entra nel trigger si elimina, contribuendo a mantenere stabile il numero di istanze attive e quindi le prestazioni complessive. Il comportamento di auto distruzione è gestito nello script Move della piattaforma.

Infine, il Prefab Obstacle rappresenta il singolo ostacolo di gioco. È costituito da un box collider e dal tag "Obstacle", che consente di rilevare la collisione con il Player e determinare la fine della partita.

L'export del gioco per il web è stato effettuato senza particolari ottimizzazioni ma semplicemente selezionando la piattaforma web tra le build per Unity. Sono state esportate due versioni e testate separatamente: una con supporto a WebGPU e WebGL e una con unico supporto a WebGL. Infine il gioco eseguibile è stato caricato su itch.io per effettuare i relativi test.

### 5.1.2 Godot

L'implementazione del prototipo Knight Runner sull'engine Godot ha seguito da vicino la struttura concettuale sviluppata in Unity, pur richiedendo una serie di adattamenti dovuti alle differenze architetturali tra i due ambienti. In particolare, è stato necessario riorganizzare parte delle componenti per aderire al modello a nodi che rappresenta la base del framework di Godot.

Per quanto concerne gli asset, sono stati adottati due formati distinti: il personaggio principale, ovvero il cavaliere, è stato importato in formato FBX, mentre i modelli relativi alle piattaforme e agli ostacoli sono stati importati in formato glb. Tale scelta si è resa necessaria a causa di alcune problematiche riscontrate nella corretta importazione delle texture attraverso FBX, che risultavano invece gestite in maniera più affidabile dal formato glb.

Questo riflette un principio metodologico più esteso: quando fattibile, è stata favorita la coerenza con Unity, mentre in presenza di incompatibilità, sono state adottate soluzioni alternative per preservare l'integrità visiva del prototipo e garantire l'uniformità del gameplay tra i diversi engine.

Passando da Unity a Godot, una delle differenze più evidenti riguarda l'organizzazione del codice. Mentre Unity permette di collegare diversi script e componenti allo stesso GameObject, in Godot ogni nodo può avere un solo script associato. Inizialmente questo può sembrare una limitazione, ma richiede semplicemente un approccio diverso: quello che prima veniva realizzato con più componenti su un singolo oggetto, ora va suddiviso in nodi distinti, ognuno con una funzione specifica.

I segnali di Godot rappresentano un'altra caratteristica interessante. Simili agli eventi di Unity ma più immediati nell'utilizzo, permettono la comunicazione tra nodi mantenendo il codice più pulito. Per la logica di gioco e la generazione procedurale del mondo, i segnali si dimostrano particolarmente efficaci, rendendo il flusso degli eventi più trasparente e modulare.

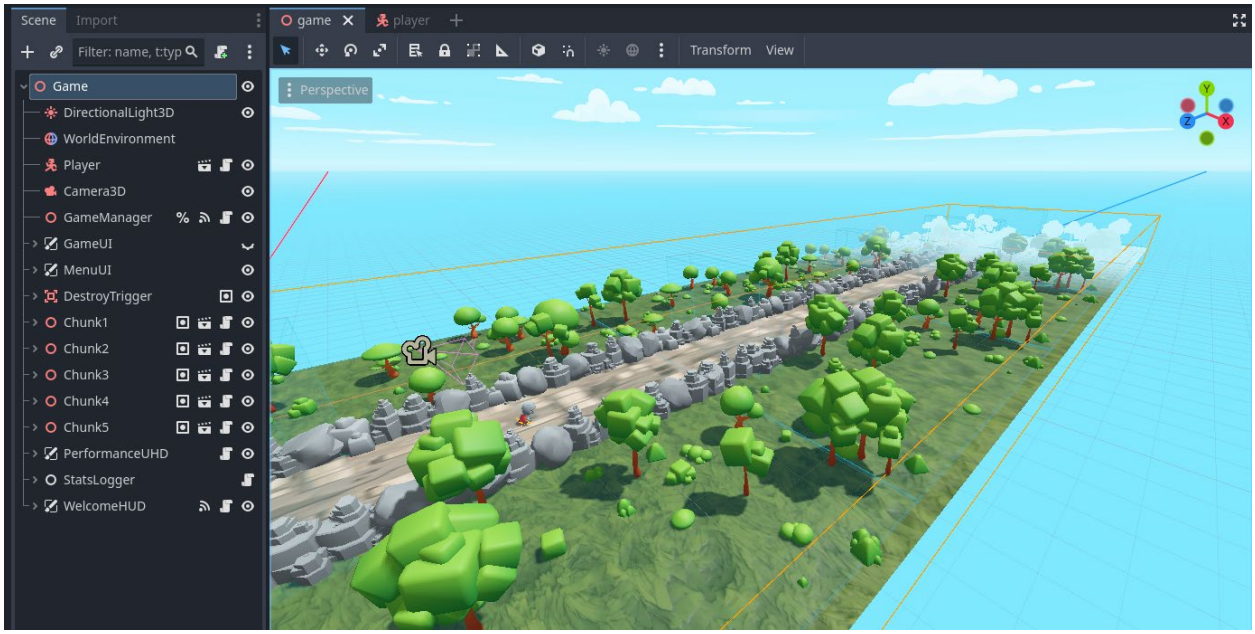



*Figura 12 - Scena principale del gioco su Godot*

L'organizzazione del progetto prevede una scena principale chiamata Game, strutturata con una gerarchia di nodi ben definita. Gli elementi principali, Player, Chunk e Obstacle, sono stati implementati come Scene separate, equivalenti ai prefab di Unity. Questa scelta mantiene l'organizzazione modulare e facilita il riutilizzo degli elementi durante la generazione procedurale.

La scena del Player ha come nodo radice un Character3D, che fornisce al personaggio la possibilità di muoversi nello spazio. A questo nodo sono collegati come figli:

- il modello 3D del cavaliere con relativo rig, responsabile della resa grafica e dell'animazione scheletrale;
- un AnimationPlayer, utilizzato per gestire le animazioni del personaggio. Si specifica che la gestione delle animazioni viene effettuata direttamente tramite script;
- una CollisionShape3D, indispensabile per abilitare le interazioni fisiche del Character3D;
- un nodo dedicato alla Trigger Street Generation, che consente la creazione procedurale dei segmenti di strada;
- un nodo Area3D, che permette di sfruttare il segnale area_entered necessario al corretto funzionamento della logica di generazione del percorso.

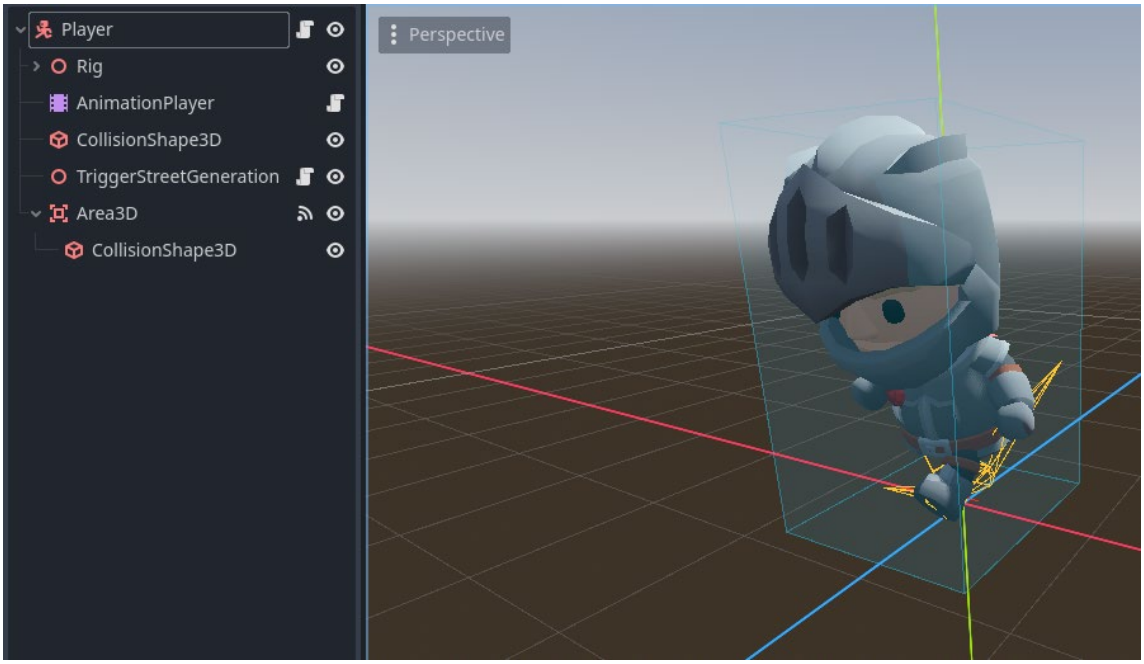



*Figura 13 - Scena del Player su Godot*

La scena Chunk è invece costruita a partire da un nodo 3D generico, al quale è associato lo script per il movimento della piattaforma. Anche in questo caso la velocità del movimento è modulata attraverso i comandi del GameManager. Ai figli del nodo principale appartengono i modelli 3D della piattaforma, un'Area3D denominata PlayerTrigger appartente al gruppo "trigger" (i gruppi sono l'equivalente dei tag in Unity), il nodo Area3d ChunkShape, che permette l'attivazione della distruzione del segmento in collisione con l'oggetto DestroyWall, e infine un nodo ObstacleSpawn, incaricato di generare ostacoli sul percorso in maniera casuale a istanziando la scena Obstacle.

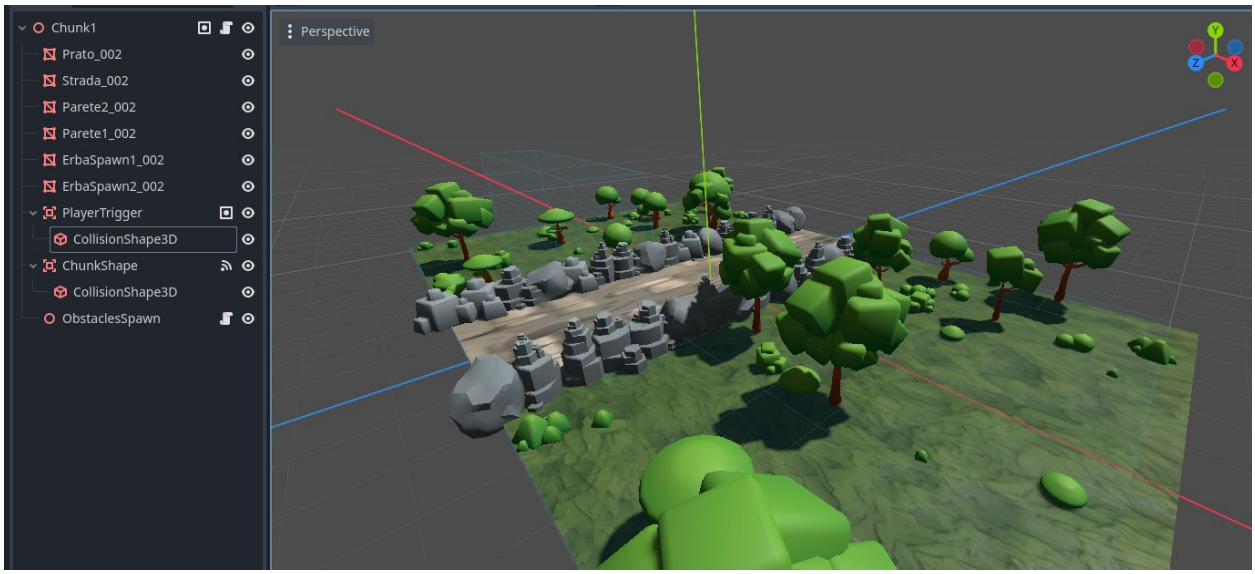



*Figura 14 - Scena del Chunk su Godot*

La scena Obstacle, dedicata alle rocce che fungono da ostacoli, è strutturata a partire da un nodo 3D di base appartenente al gruppo “obstacles”. Come figli, essa include i modelli grafici che rappresentano l’ostacolo e un’Area3D con box collider, anch’essa assegnata al gruppo “obstacles”, responsabile della gestione delle collisioni.

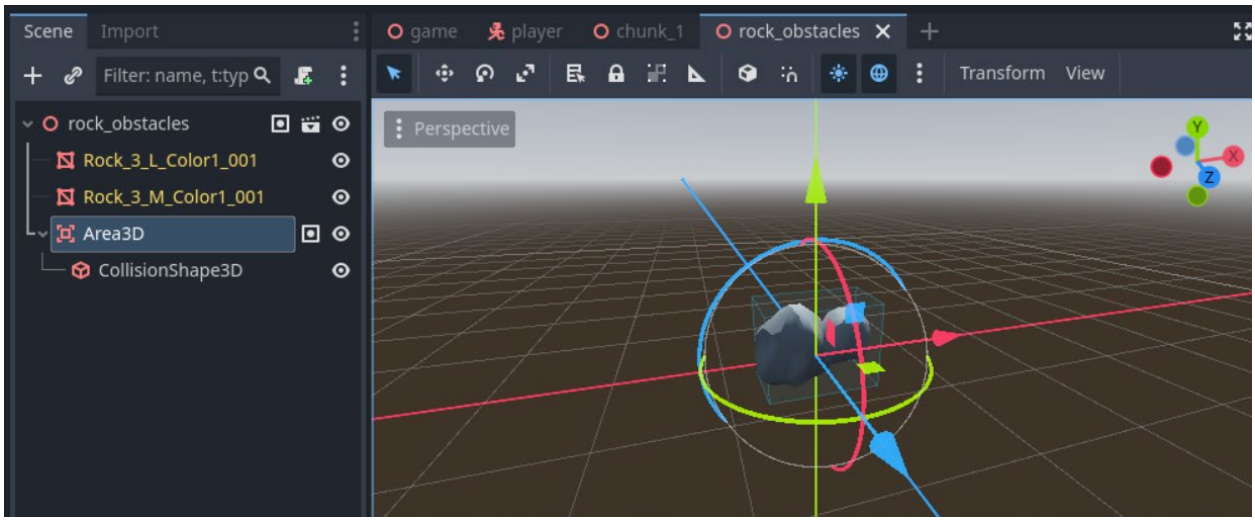



*Figura 15 - Scena dell'ostacolo su Godot*

Il GameManager di Godot è stato sviluppato con un funzionamento del tutto analogo a quello realizzato in Unity, con la differenza che, anziché sfruttare eventi personalizzati, fa uso dei segnali per coordinare il flusso di gioco. Questo nodo, identificato con un nome univoco per facilitarne il richiamo da altri componenti, gestisce la transizione tra stati di gioco (menu, partita attiva, game over), regola la velocità delle piattaforme e aggiorna il punteggio visualizzato dall’utente.

L’interfaccia utente è stata realizzata tramite i nodi UI messi a disposizione nativamente da Godot.

La logica alla base della raccolta dati è rimasta sostanzialmente invariata rispetto a Unity: anche in questo caso i dati prestazionali vengono raccolti in tempo reale, impacchettati in un payload JSON e inviati mediante una richiesta HTTP (HttpRequest) a un server esterno, simulando il comportamento di un form web. Questa coerenza metodologica ha permesso di confrontare direttamente i dati raccolti nei diversi engine, senza introdurre bias dovuti a differenze nell'implementazione.

Per quanto riguarda la build finale, questa è stata generata utilizzando il preset Web fornito nativamente da Godot, adottando come modalità di rendering il profilo Forward+. Tale scelta è stata dettata dall'esigenza di garantire un equilibrio tra qualità visiva e prestazioni, dal momento che Forward+ rappresenta un compromesso efficace per applicazioni interattive 3D distribuite via browser, in grado di supportare illuminazione dinamica mantenendo una buona efficienza nei calcoli grafici.

Un aspetto peculiare dell'implementazione in Godot è stato lo sviluppo di due differenti build del prototipo, realizzate in versioni diverse dell'engine. Al momento dello sviluppo, infatti, la versione stabile disponibile era la 4.4, che ha costituito il punto di riferimento principale. Tuttavia, sulla base di quanto riportato sia nella documentazione ufficiale sia negli articoli pubblicati sul blog di Godot, la successiva versione 4.5 (beta 5) introduceva ottimizzazioni prestazionali di default, applicabili out of the box senza interventi aggiuntivi da parte dello sviluppatore. Per verificare l'impatto effettivo di tali miglioramenti, si è quindi deciso di produrre anche una seconda build del prototipo con la versione più recente.

Questa scelta metodologica consente non solo di garantire una valutazione più completa delle potenzialità dell'engine, ma anche di osservare in maniera diretta come l'evoluzione del software possa incidere sulle prestazioni finali di un progetto distribuito sul web.

L'implementazione in Godot ha richiesto una maggiore frammentazione degli oggetti rispetto a Unity, dovuta alla natura gerarchica dei nodi, ma ha beneficiato del sistema dei segnali che ha reso il flusso logico più modulare e leggibile. Pur con alcune difficoltà legate all'importazione degli asset e alla necessità di gestire manualmente determinati collegamenti, l'engine si è dimostrato adatto a replicare in maniera fedele la logica del prototipo, mantenendo al contempo la comparabilità con le altre versioni sviluppate.

### 5.1.3 Playcanvas

L'implementazione del prototipo in PlayCanvas si è rivelata, sotto molti aspetti, più vicina all'esperienza di sviluppo in Unity rispetto a quella in Godot. Questo è dovuto al fatto che l'engine, pur essendo interamente basato sul web e accessibile tramite un editor online, presenta concetti e strutture familiari per chi proviene da ambienti come Unity, in particolare grazie alla presenza di prefab e di un sistema di eventi globali. Tali caratteristiche hanno reso il passaggio da Unity a PlayCanvas relativamente naturale e poco problematico. Dal punto di vista del workflow, l'editor di PlayCanvas offre strumenti integrati e funzionalità collaborative tipiche di un ambiente "cloud–based", ma al tempo stesso permette una certa continuità con i flussi di lavoro offline. Un aspetto particolarmente rilevante è la disponibilità di un'estensione per Visual Studio Code, che consente di sincronizzare l'intera directory del progetto e di modificare gli script direttamente dall'editor desktop. In questo modo, lo sviluppatore mantiene un accesso immediato ai file e può lavorare con la stessa fluidità di un engine tradizionale, riducendo notevolmente l'impatto del vincolo di dover operare in un editor online.

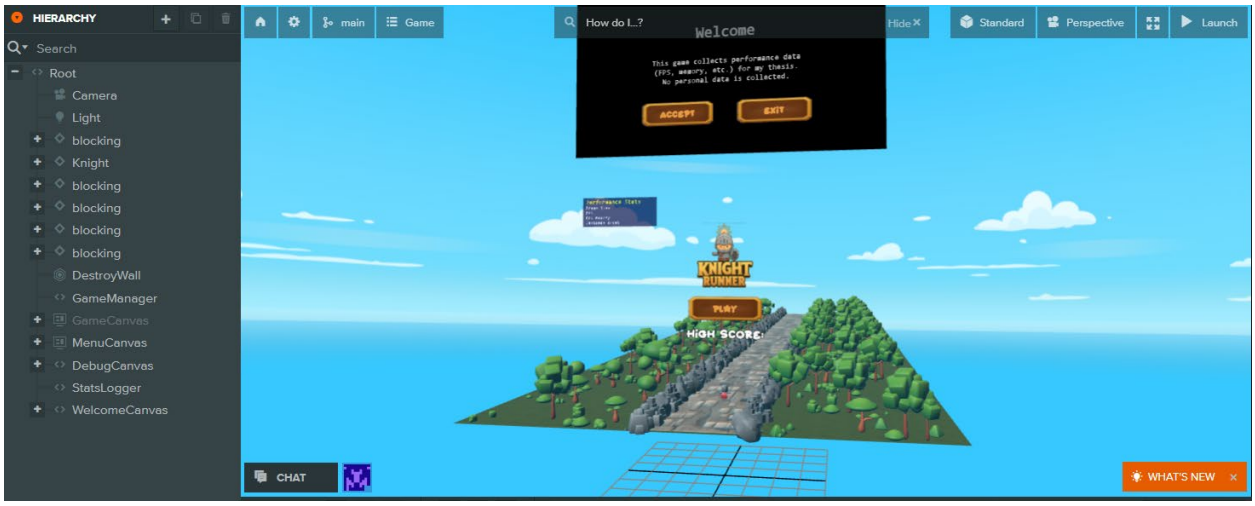



*Figura 16 - Scena principale del gioco su Playcanvas*

Dal punto di vista tecnico, PlayCanvas ha mostrato un'ottima compatibilità con i file in formato glTF/glb, che sono stati utilizzati per importare sia i modelli statici sia quelli animati. La gestione delle animazioni avviene attraverso un componente dedicato e tramite uno state graph, soluzione che richiama molto da vicino l'Animator Controller di Unity. Questo approccio ha reso la migrazione delle logiche di animazione particolarmente lineare, evitando difficoltà legate a differenze sintattiche o concettuali.

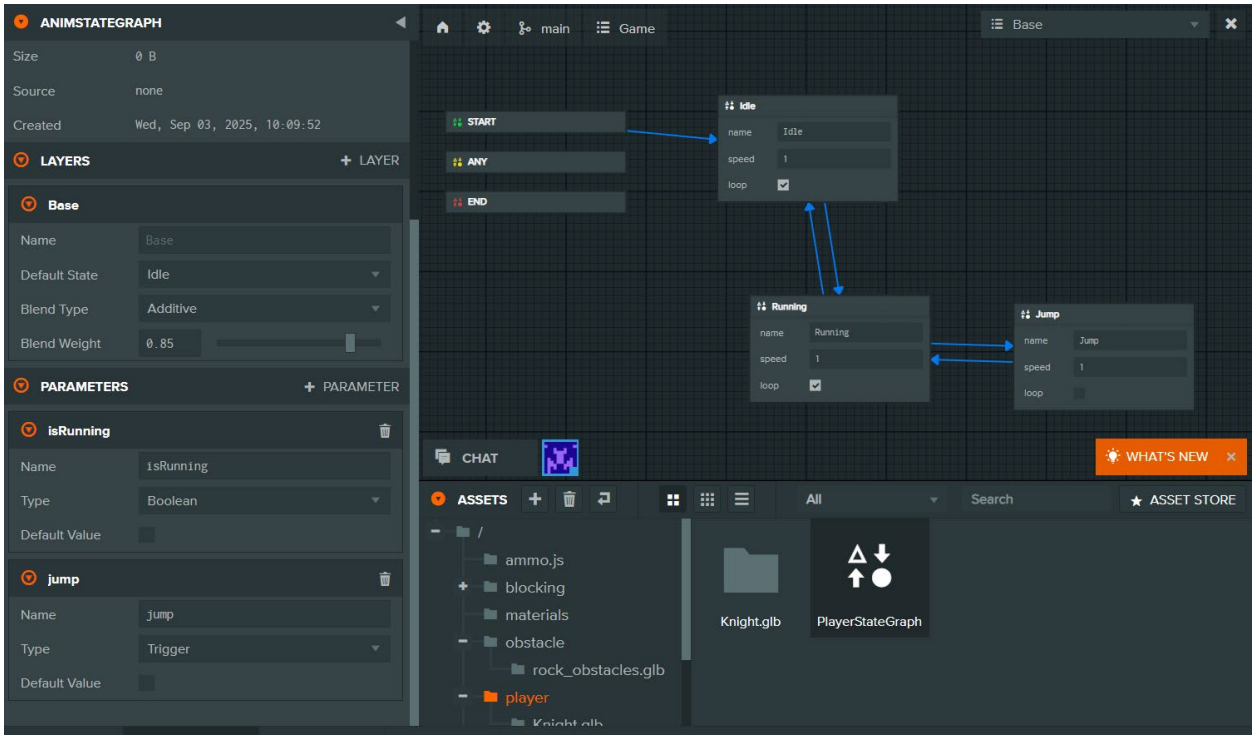



*Figura 17 - Grafo delle animazioni utilizzato in Playcanvas*

In ambito fisico, PlayCanvas non dispone di un motore proprietario ma si appoggia alla libreria esterna ammo.js, una trasposizione in JavaScript del noto motore Bullet Physics. L'impiego di tale libreria, sebbene fornisca un fondamento solido per la gestione delle collisioni e della dinamica, ha messo in luce alcune limitazioni, specialmente su dispositivi meno performanti. In particolare, è stato riscontrato un problema nella rilevazione delle collisioni: quando gli oggetti si trovavano nella posizione (0,0,0), talvolta venivano registrate collisioni inesistenti, introducendo

comportamenti anomali. Questo aspetto ha reso necessario un controllo più rigoroso nella gestione delle posizioni iniziali degli oggetti per ridurre il verificarsi di falsi positivi.

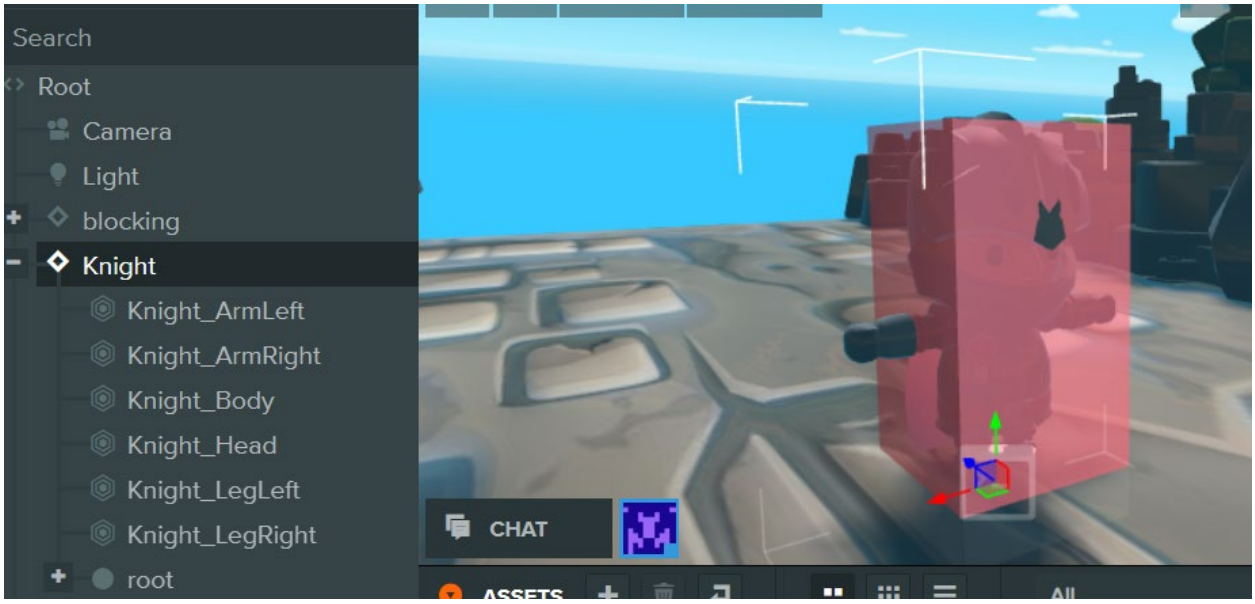



*Figura 18 - Focus sul prefab del Player in Playcanvas*

Un'ulteriore peculiarità riguarda la gestione dei trigger, che in PlayCanvas segue una logica differente da quella di Unity. Nel motore di Unity, un trigger richiede la presenza di un collider contrassegnato come "is trigger" e di un rigidbody per poter generare eventi di collisione.

In PlayCanvas, invece, è sufficiente assegnare a un oggetto un componente di collisione, senza necessità di un rigidbody. Solo il componente in questione è inoltre in grado di rilevare l'evento di trigger. Tale differenza, in apparenza trascurabile, ha richiesto un leggero adattamento nella logica del prototipo: nel caso del giocatore che deve attraversare il trigger della piattaforma, l'evento di trigger viene prima attivato dal componente associato al segmento di strada e solo successivamente intercettato dal Player, il quale, mediante lo script Trigger Street Generation, provvede alla generazione di nuove piattaforme.

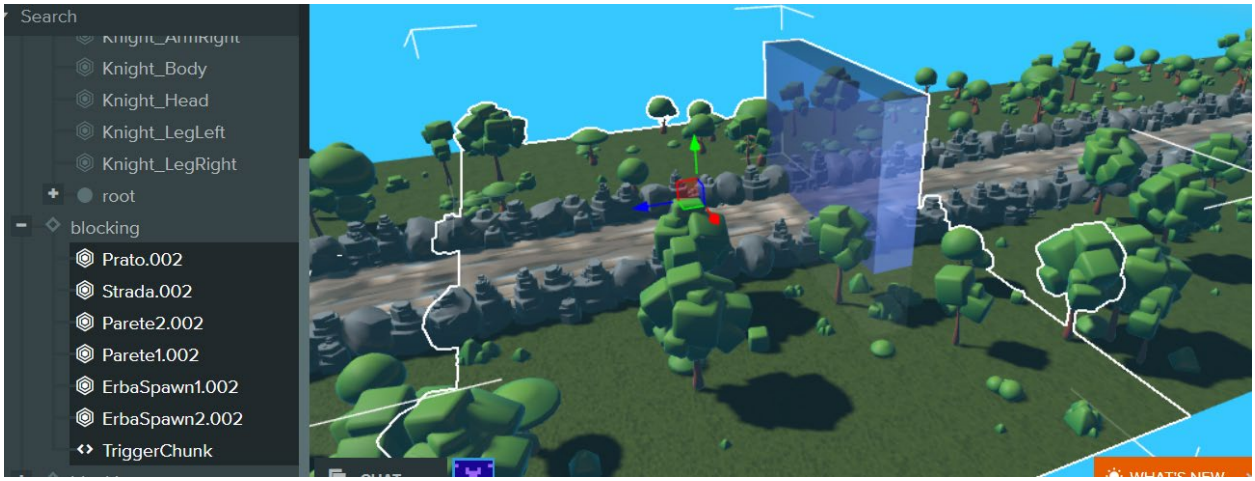



*Figura 19 - Focus sul pezzo di strada in Playcanvas*

L'implementazione della UI non ha richiesto particolari adattamenti, in quanto PlayCanvas fornisce un insieme di componenti dedicati che consentono la realizzazione di interfacce grafiche in modo diretto e intuitivo, mantenendo un approccio simile a quello già sperimentato in Unity. Analogamente, la logica di raccolta e trasmissione dei dati prestazionali è stata replicata in modo

coerente con gli altri prototipi: i parametri sono stati raccolti in tempo reale e inviati periodicamente al server remoto tramite richieste HTTP, garantendo la comparabilità dei risultati sperimentali.

Inoltre, un elemento distintivo di PlayCanvas riguarda le opzioni di deployment. In quanto motore nativamente web-based, l'engine consente non solo di esportare un progetto in formato web tradizionale, ma anche di pubblicarlo direttamente sulla piattaforma PlayCanvas, semplificando notevolmente la distribuzione. Inoltre, lo sviluppatore ha la possibilità di selezionare se generare una build basata su WebGL, su WebGPU o su entrambe le tecnologie. Tale possibilità rappresenta un vantaggio significativo in termini di sperimentazione e compatibilità, in quanto consente di valutare direttamente le differenze prestazionali derivanti dall'uso delle due API grafiche.

Lo sviluppo del prototipo in PlayCanvas si è rivelato complessivamente scorrevole con piccoli ostacoli soprattutto riguardanti il problema della fisica discusso in precedenza. La combinazione tra un editor cloud collaborativo, la possibilità di integrazione con strumenti desktop e la presenza di feature concettualmente simili a Unity ha reso l’esperienza di sviluppo particolarmente accessibile, pur con alcune limitazioni nella fisica e nella gestione dei trigger. L’engine si è dimostrato una piattaforma matura per la realizzazione di giochi 3D direttamente fruibili via browser, evidenziando un potenziale interessante soprattutto in scenari in cui la rapidità del deploy rappresenta un fattore cruciale.

### 5.1.4 Rogue Engine

L’implementazione del prototipo in Rogue Engine, un editor e framework relativamente recente basato su Three.js, ha evidenziato caratteristiche e criticità peculiari rispetto agli altri motori considerati.

A differenza di Unity, Godot o PlayCanvas, che possono contare su comunità consolidate e una documentazione estesa, Rogue Engine si presenta come una piattaforma ancora in fase embrionale, con un ecosistema meno strutturato e un livello di stabilità non sempre adeguato a progetti complessi.

Una prima difficoltà riscontrata riguarda proprio l’editor che, pur ispirandosi fortemente all’interfaccia di Unity, riprendendo la logica dei prefab e dei componenti multipli applicabili a ciascun game object, risulta in parte acerbo e instabile.

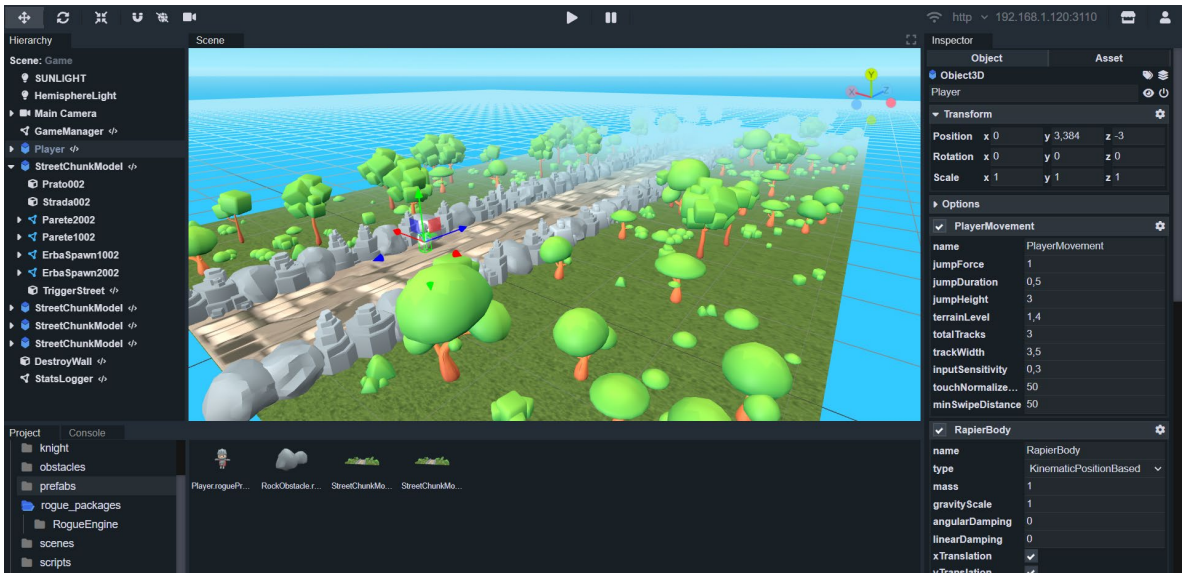



*Figura 20 - Scena principale del gioco su Rogue Engine*

Numerose funzionalità considerate basilari negli engine più maturi (come la gestione delle animazioni o della fisica) non sono integrate nativamente, ma devono essere aggiunte tramite un marketplace di estensioni. Questo approccio, se da un lato garantisce modularità e personalizzazione, dall'altro rende più frammentaria l'esperienza di sviluppo, poiché la qualità e la stabilità delle estensioni variano in base al contributo dei singoli sviluppatori.

Dal punto di vista della compatibilità degli asset, si è optato per il formato glb, che ha garantito un'integrazione più fluida con l'engine. L'importazione dei modelli non ha presentato particolari problematiche, tuttavia la gestione successiva, in termini di animazioni e fisica, ha richiesto soluzioni specifiche.

La gestione delle animazioni è stata realizzata tramite l'estensione Rogue Animator, che offre un sistema molto più semplificato rispetto a quello di Unity o PlayCanvas. In assenza di un vero e proprio animation tree, le animazioni vengono associate direttamente a una chiave, con un'assegnazione che avviene tramite editor, con la possibilità di richiamarle via script ed effettuare operazioni di blending manualmente via codice. Questo approccio si è rivelato poco pratico e macchinoso nella realizzazione del prototipo, richiedendo un maggior sforzo di scripting e una gestione meno intuitiva delle transizioni tra stati animati.

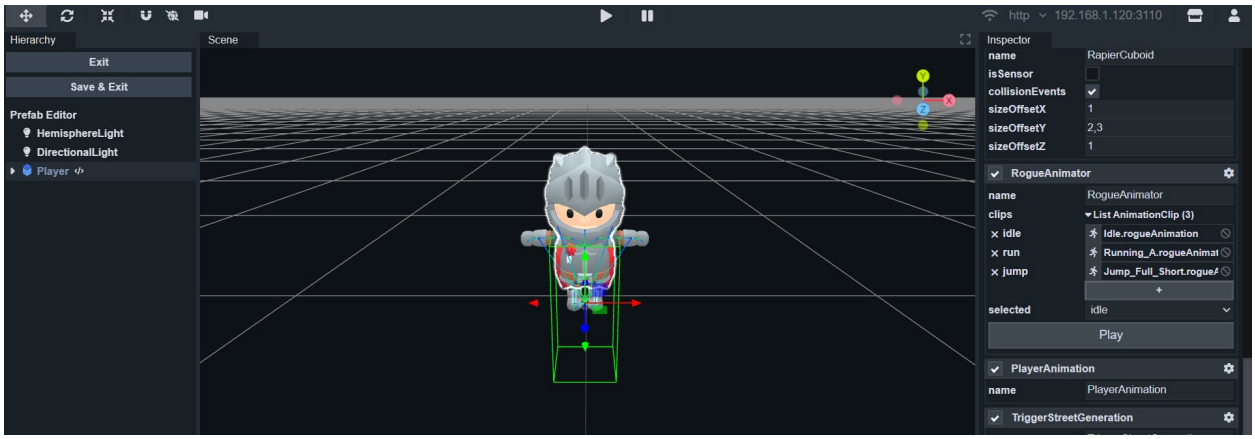



*Figura 21 - Visualizzazione del prefab del Player su Rogue Engine*

Per quanto riguarda la fisica, è stata utilizzata l'estensione Rogue Rapier, basata sul motore Rapier. Anche in questo caso, l'implementazione ha evidenziato alcune criticità, in particolare

nella configurazione e modifica dei collider, che non sempre venivano riconosciuti correttamente. La soluzione a tali problematiche ha richiesto l'adozione di "trucchi" legati alla gerarchia degli oggetti, come l'uso del parenting per forzare determinati comportamenti.

La creazione dei trigger si è rivelata altrettanto complessa: in assenza di una documentazione dettagliata, è stato necessario procedere per tentativi, sperimentando diverse combinazioni di parametri sul RapierBody, fino ad ottenere un comportamento soddisfacente. Sebbene il risultato finale abbia permesso di replicare le dinamiche di gioco previste, il processo è stato decisamente meno lineare rispetto agli altri engine.

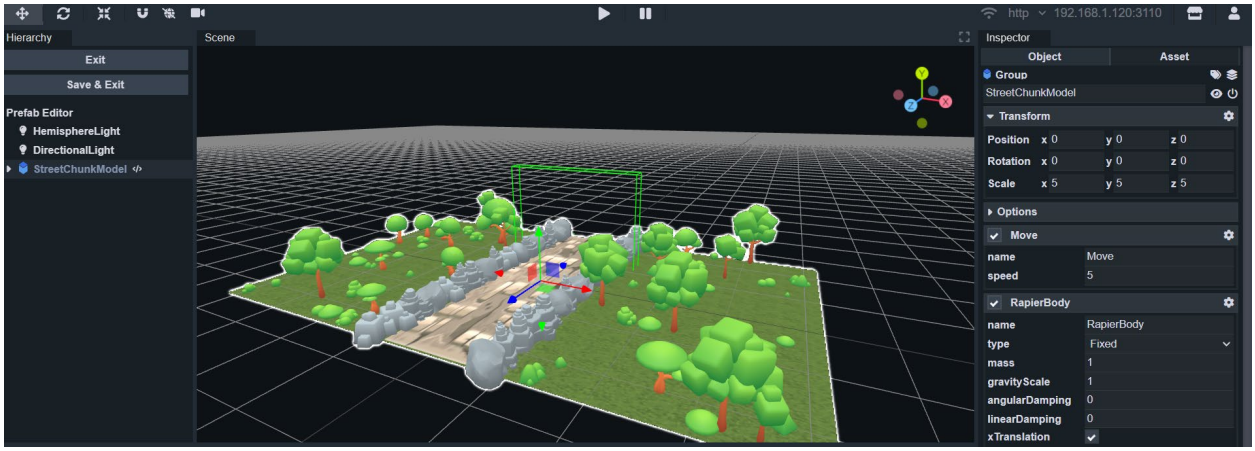



*Figura 22 - Visualizzazione del prefab della strada su Rogue Engine*

Un aspetto peculiare di Rogue Engine è rappresentato dall'implementazione della UI, che non si basa su un sistema interno come nei casi di Unity, Godot o PlayCanvas, ma sfrutta direttamente codice HTML e CSS sovrapposto al canvas di rendering 3D. Questo approccio, se da un lato apre a una grande flessibilità nell'integrazione di interfacce grafiche web-native, dall'altro si discosta notevolmente dalle prassi tipiche dei game engine, imponendo allo sviluppatore di combinare logiche di sviluppo di videogiochi con quelle del web design tradizionale. Tale scelta si è rivelata interessante dal punto di vista sperimentale, ma meno pratica per un flusso di lavoro unificato, soprattutto in un contesto in cui l'obiettivo era garantire comparabilità tra diversi engine.

Il processo di build si è invece dimostrato particolarmente semplice e diretto. Una volta selezionata la scena da esportare, l'engine procede autonomamente alla generazione della build, senza offrire all'utente un controllo avanzato sui parametri di configurazione. Questa immediatezza, sebbene positiva in termini di rapidità, limita tuttavia la possibilità di effettuare ottimizzazioni mirate o personalizzazioni avanzate del processo di deploy.

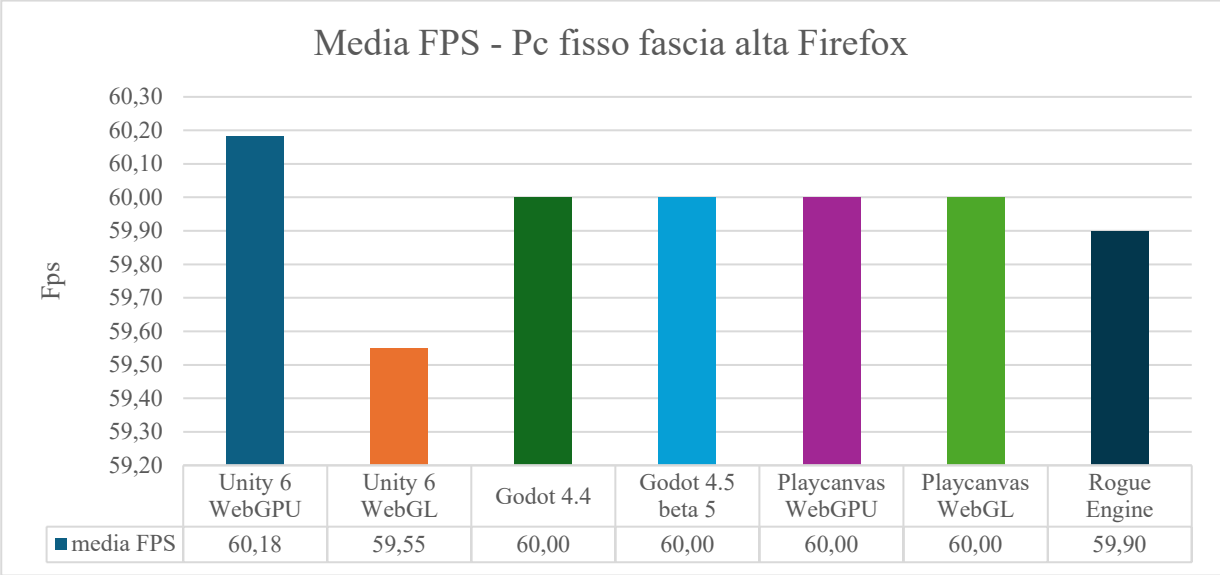
L'esperienza con Rogue Engine ha messo in luce i limiti di un ambiente ancora giovane, che si affida pesantemente a estensioni esterne per coprire funzionalità di base e che risente di una documentazione scarsa, affidata quasi esclusivamente a tutorial rilasciati dallo stesso creatore. Ciò nonostante, l'engine rappresenta un'interessante alternativa per lo sviluppo web-native basato su Three.js, offrendo una struttura concettuale familiare a chi proviene da Unity e un approccio fortemente integrato con tecnologie web standard. Le difficoltà incontrate, soprattutto nella fisica e nelle animazioni, evidenziano tuttavia come, allo stato attuale, Rogue Engine sia meno competitivo rispetto ad altri ambienti più maturi, almeno in scenari in cui stabilità e rapidità di implementazione risultano cruciali.

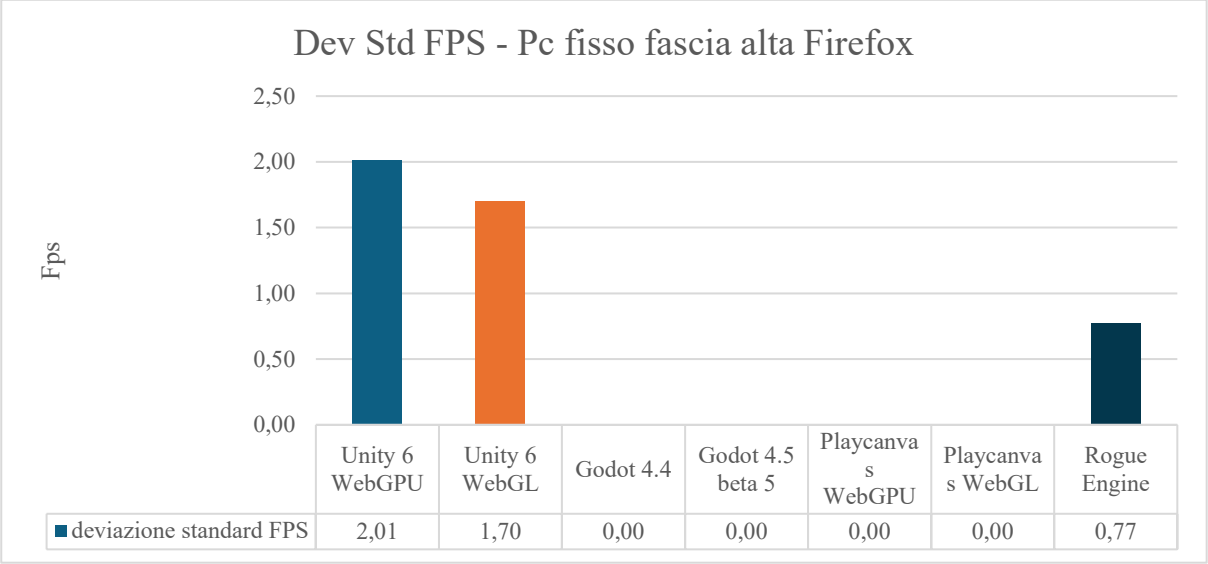
# 6. Risultati

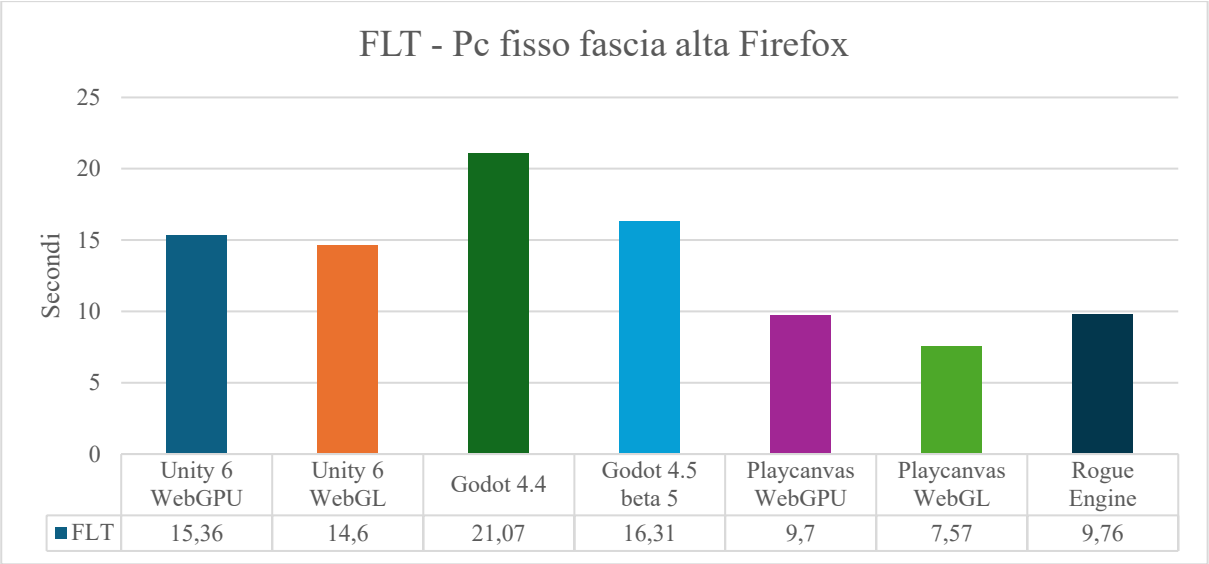
Di seguito si elencano i risultati ottenuti dagli esperimenti sintetizzati in tabelle e grafici.

Pc windows fisso fascia alta, test effettuato con browser Firefox:

| | Unity 6 WebGPU | Unity 6 WebGL | Godot 4.4 | Godot 4.5 beta 5 | Playcanvas WebGPU | Playcanvas WebGL | Rogue Engine |
|---|---|---|---|---|---|---|---|
| media FPS | 60,18 | 59,55 | 60,00 | 60,00 | 60,00 | 60,00 | 59,90 |
| media frame time | 16.38 | 16.49 | 17.07 | 17.07 | 17,08 | 17,08 | 16,69 |
| minimo FPS | 56 | 56 | 60 | 60 | 60 | 60 | 54 |
| massimo FPS | 63 | 62 | 60 | 60 | 60 | 60 | 60 |
| deviazione standard FPS | 2,01 | 1,70 | 0,00 | 0,00 | 0,00 | 0,00 | 0,77 |
| FLT | 15,36 | 14,6 | 21,07 | 16,31 | 9,7 | 7,57 | 9,76 |

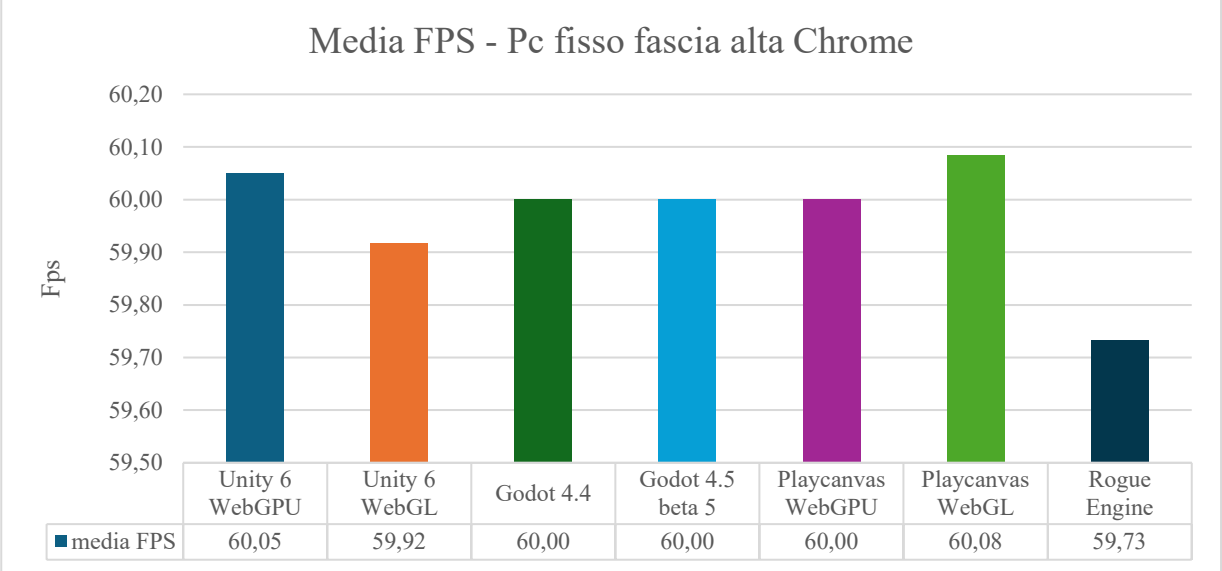


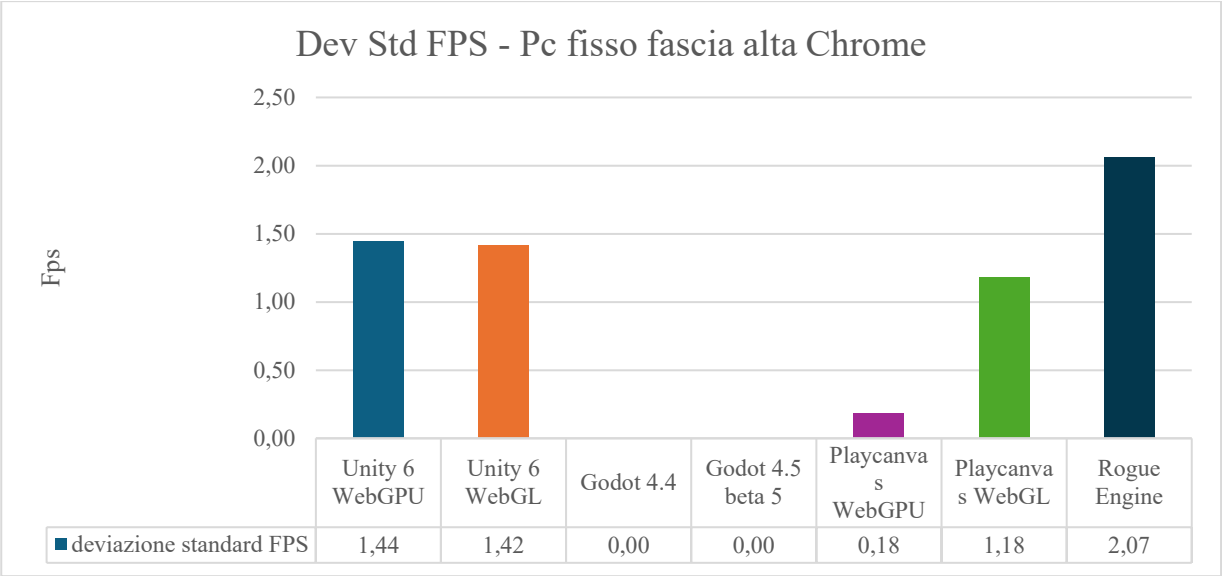



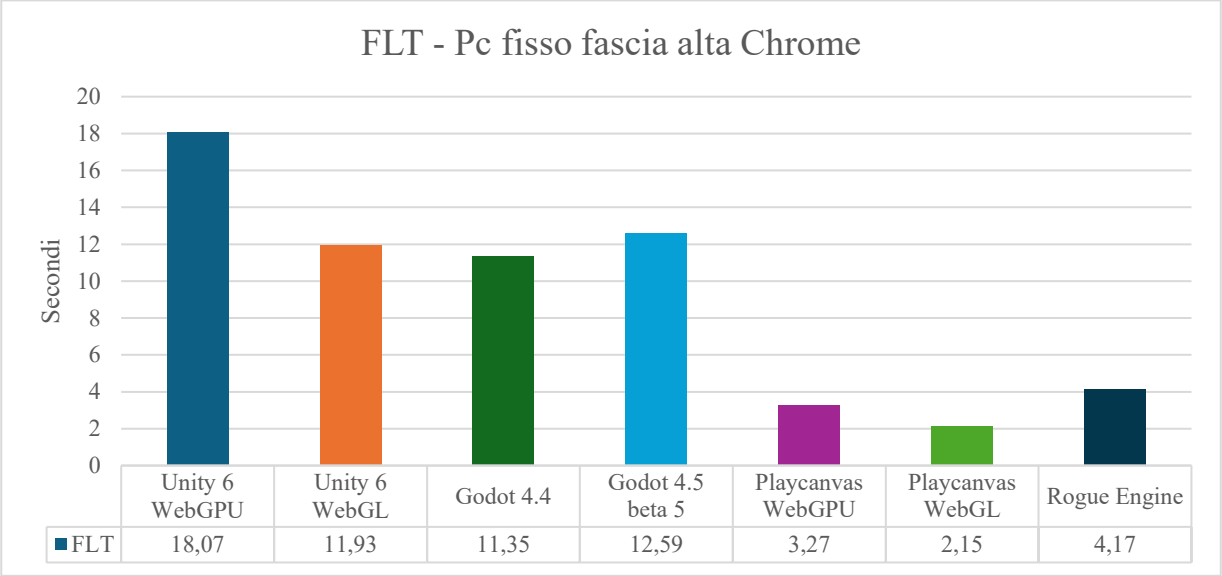



Pc windows fisso fascia alta, test effettuato con browser Chrome:

| | Unity 6 WebGPU | Unity 6 WebGL | Godot 4.4 | Godot 4.5 beta 5 | Playcanvas WebGPU | Playcanvas WebGL | Rogue Engine |
|---|---|---|---|---|---|---|---|
| media FPS | 60,05 | 59,92 | 60,00 | 60,00 | 60,00 | 60,08 | 59,73 |
| media frame time | 16.39 | 16.42 | 17.07 | 17.07 | 17,08 | 17,03 | 16,77 |
| minimo FPS | 59 | 59 | 60 | 60 | 59 | 57 | 44 |
| massimo FPS | 62 | 63 | 60 | 60 | 61 | 68 | 60 |
| deviazione standard FPS | 1,44 | 1,42 | 0,00 | 0,00 | 0,18 | 1,18 | 2,07 |
| FLT | 18,07 | 11,93 | 11,35 | 12,59 | 3,27 | 2,15 | 4,17 |

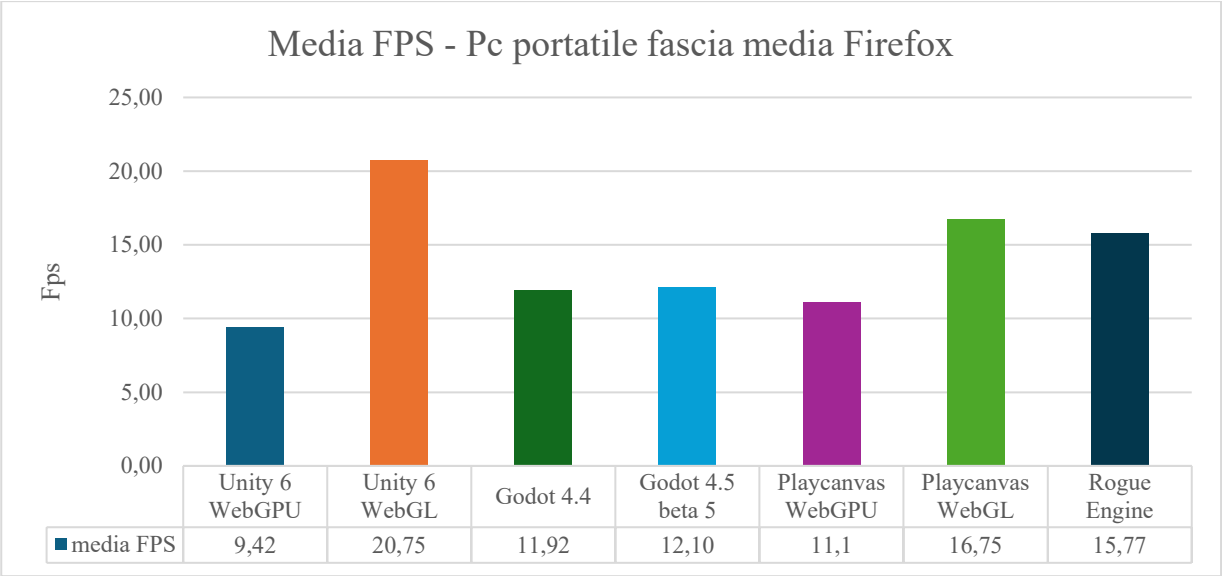


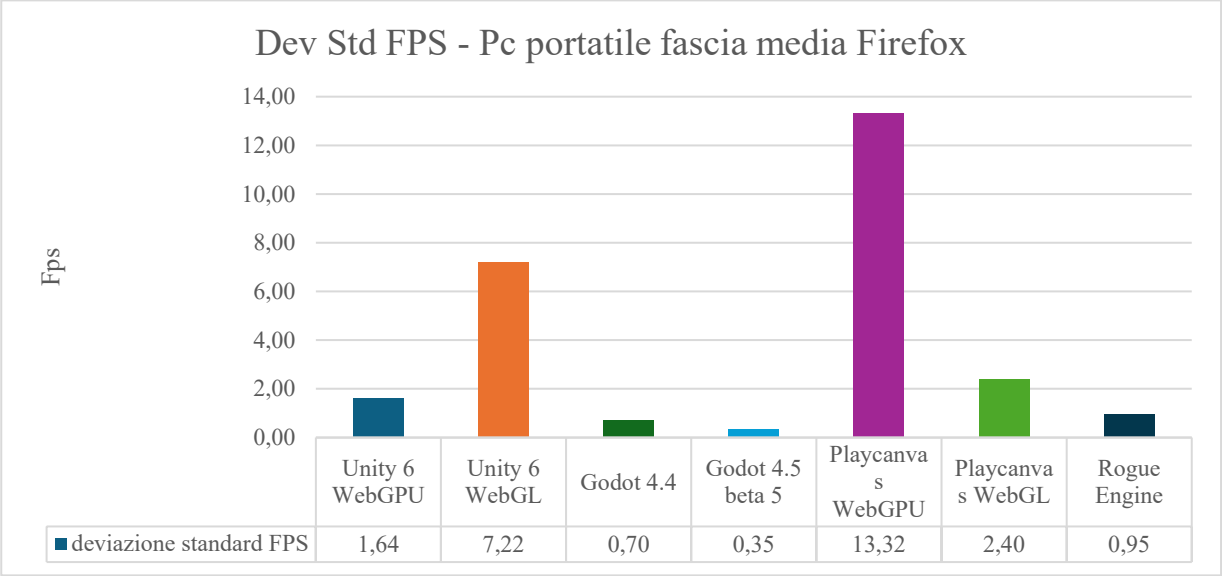


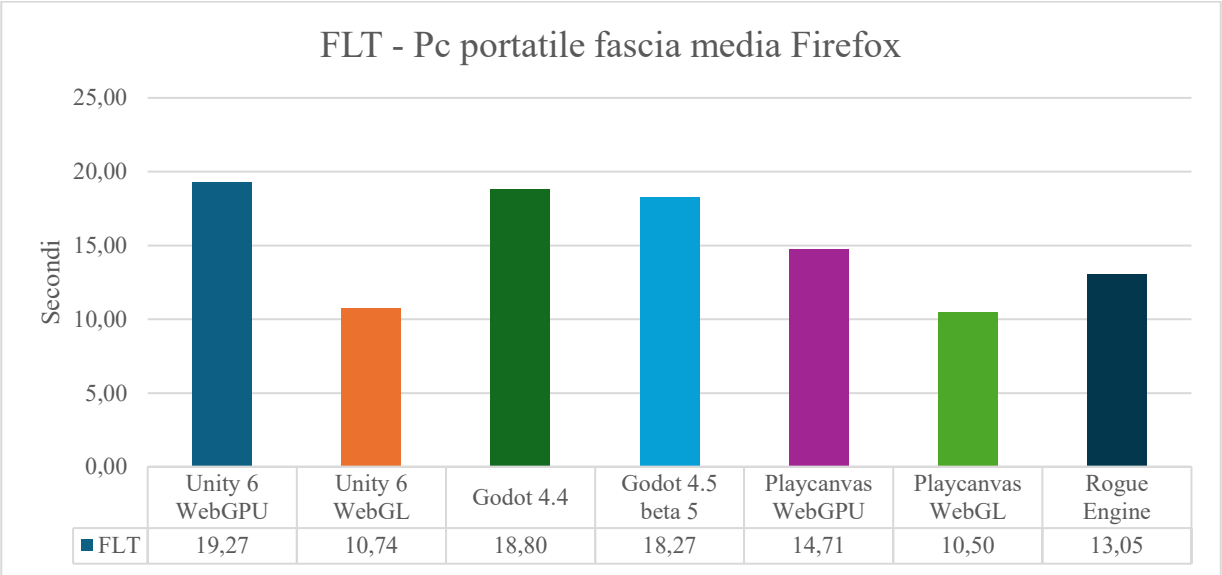




Pc portatile fascia media, test effettuato con browser Firefox:

| | Unity 6 WebGPU | Unity 6 WebGL | Godot 4.4 | Godot 4.5 beta 5 | Playcanvas WebGPU | Playcanvas WebGL | Rogue Engine |
|---|---|---|---|---|---|---|---|
| media FPS | 9,42 | 20,75 | 11,92 | 12,10 | 11,1 | 16,75 | 15,77 |
| media frame time | 106,83 | 51,75 | 84,26 | 82,76 | 137,36 | 61,14 | 64,76 |
| minimo FPS | 9 | 12 | 11 | 11 | 4 | 15 | 9 |
| massimo FPS | 19 | 50 | 15 | 13 | 60 | 20 | 16 |
| deviazione standard FPS | 1,64 | 7,22 | 0,70 | 0,35 | 13,32 | 2,40 | 0,95 |
| FLT | 19,27 | 10,74 | 18,80 | 18,27 | 14,71 | 10,50 | 13,05 |

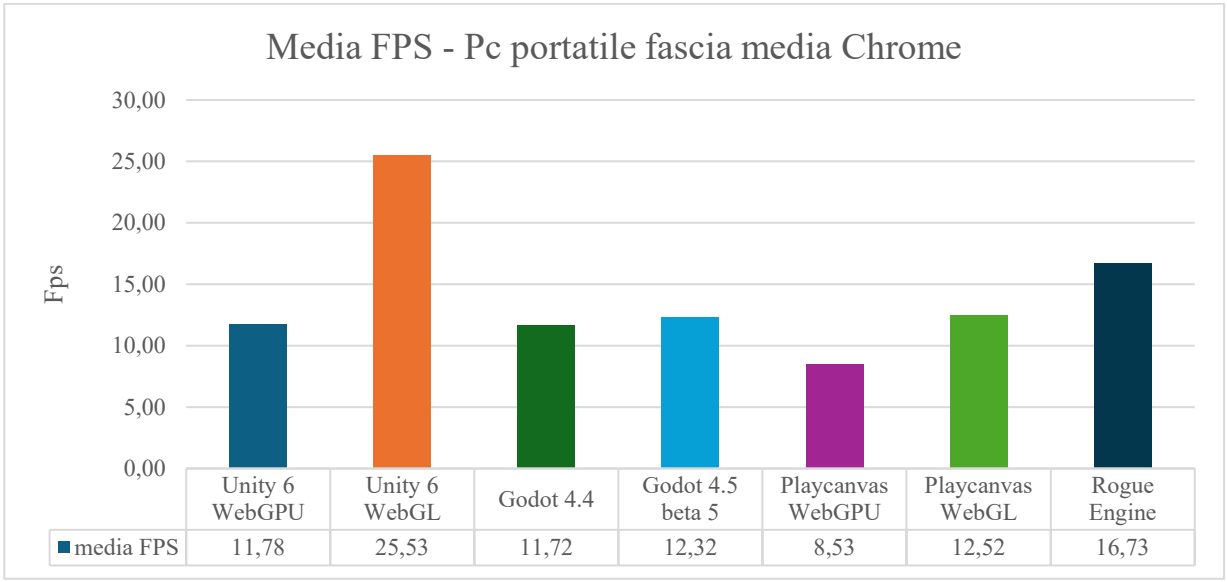

Media FPS - Pc portatile fascia media Firefox
Fps
25,00
20,00
15,00
10,00
5,00
0,00
Unity 6 WebGPU
Unity 6 WebGL
Godot 4.4
Godot 4.5 beta 5
Playcanvas WebGPU
Playcanvas WebGL
Rogue Engine
media FPS
9,42
20,75
11,92
12,10
11,1
16,75
15,77

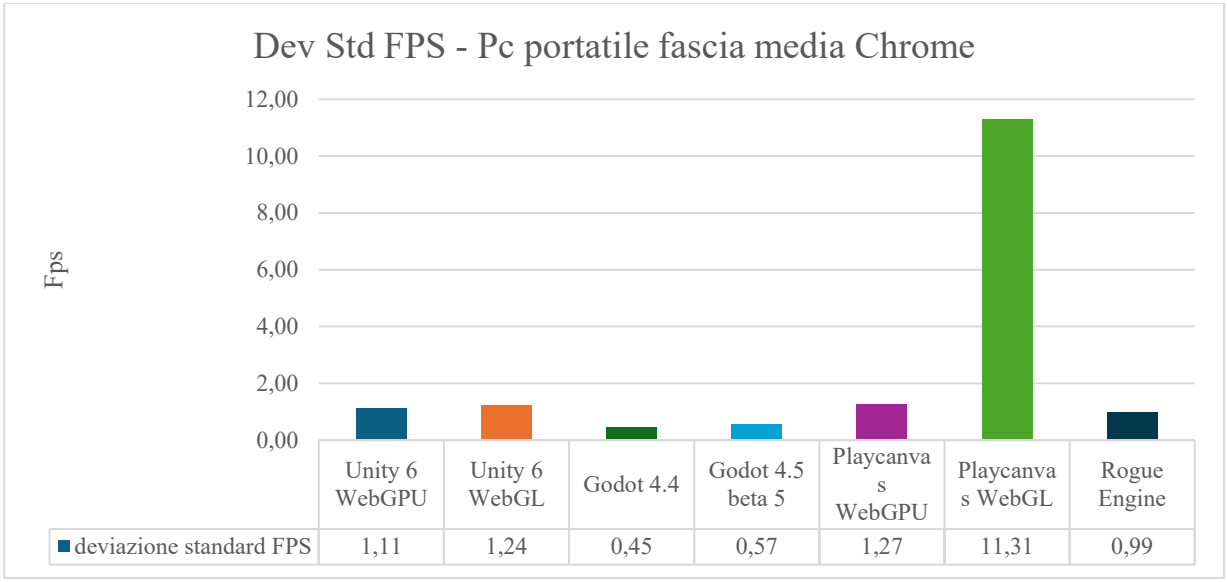

Dev Std FPS - Pc portatile fascia media Firefox
Fps
14,00
12,00
10,00
8,00
6,00
4,00
2,00
0,00
Unity 6 WebGPU
Unity 6 WebGL
Godot 4.4
Godot 4.5 beta 5
Playcanvas WebGPU
Playcanvas WebGL
Rogue Engine
deviazione standard FPS
1,64
7,22
0,70
0,35
13,32
2,40
0,95

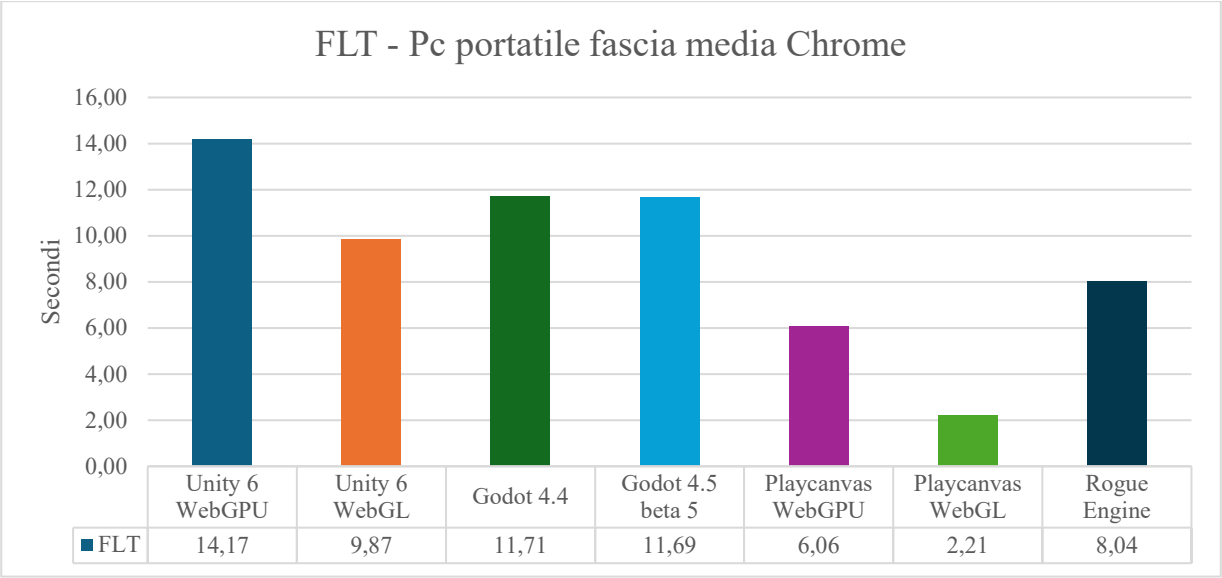

FLT - Pc portatile fascia media Firefox
Secondi
25,00
20,00
15,00
10,00
5,00
0,00
Unity 6 WebGPU
Unity 6 WebGL
Godot 4.4
Godot 4.5 beta 5
Playcanvas WebGPU
Playcanvas WebGL
Rogue Engine
FLT
19,27
10,74
18,80
18,27
14,71
10,50
13,05

Pc portatile fascia media, test effettuato con browser Chrome:

| | Unity 6 WebGPU | Unity 6 WebGL | Godot 4.4 | Godot 4.5 beta 5 | Playcanvas WebGPU | Playcanvas WebGL | Rogue Engine |
|---|---|---|---|---|---|---|---|
| media FPS | 11,78 | 25,53 | 11,72 | 12,32 | 8,53 | 12,52 | 16,73 |
| media frame time | 84,92 | 39,20 | 85,59 | 81,53 | 122,96 | 102,68 | 59,80 |
| minimo FPS | 9 | 23 | 11 | 11 | 6 | 5 | 14 |
| massimo FPS | 15 | 29 | 12 | 13 | 12 | 60 | 19 |
| deviazione standard FPS | 1,11 | 1,24 | 0,45 | 0,57 | 1,27 | 11,31 | 0,99 |
| FLT | 14,17 | 9,87 | 11,71 | 11,69 | 6,06 | 2,21 | 8,04 |

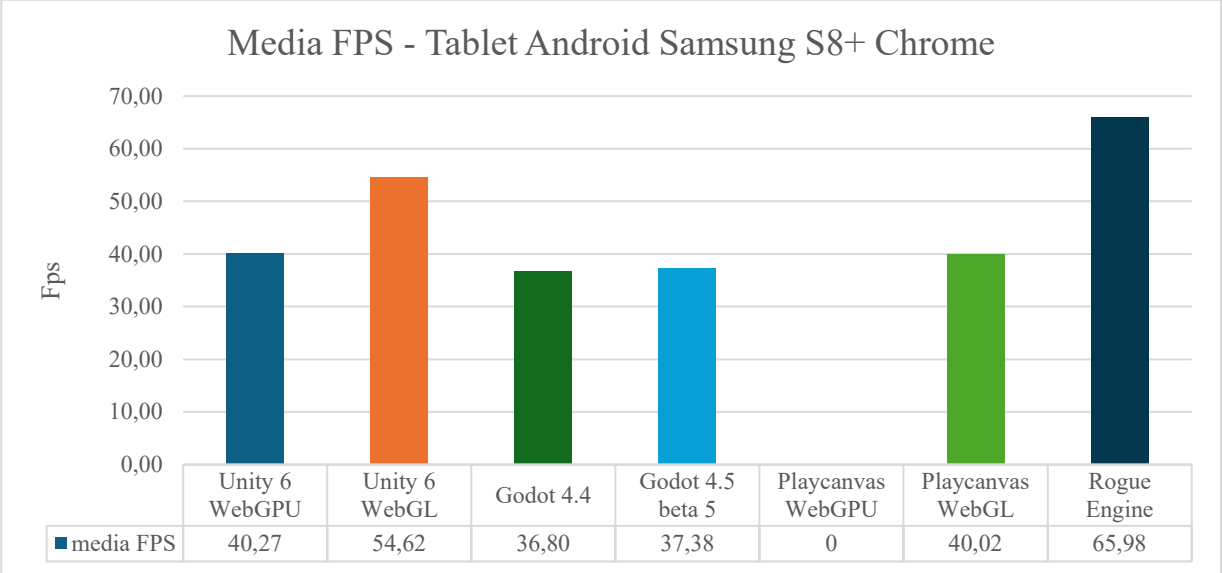


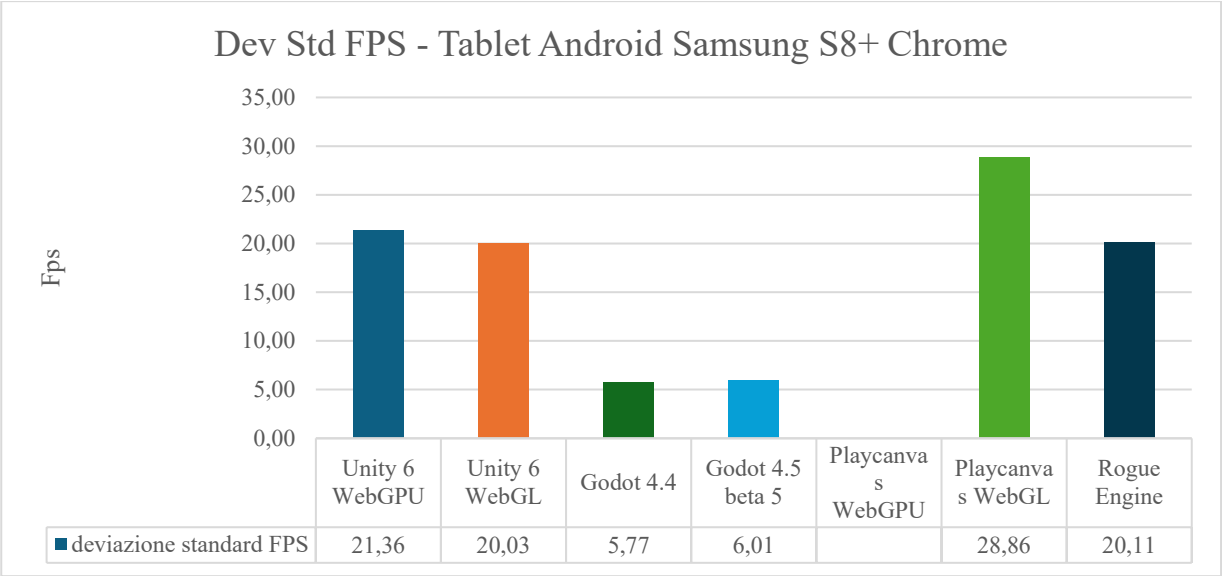



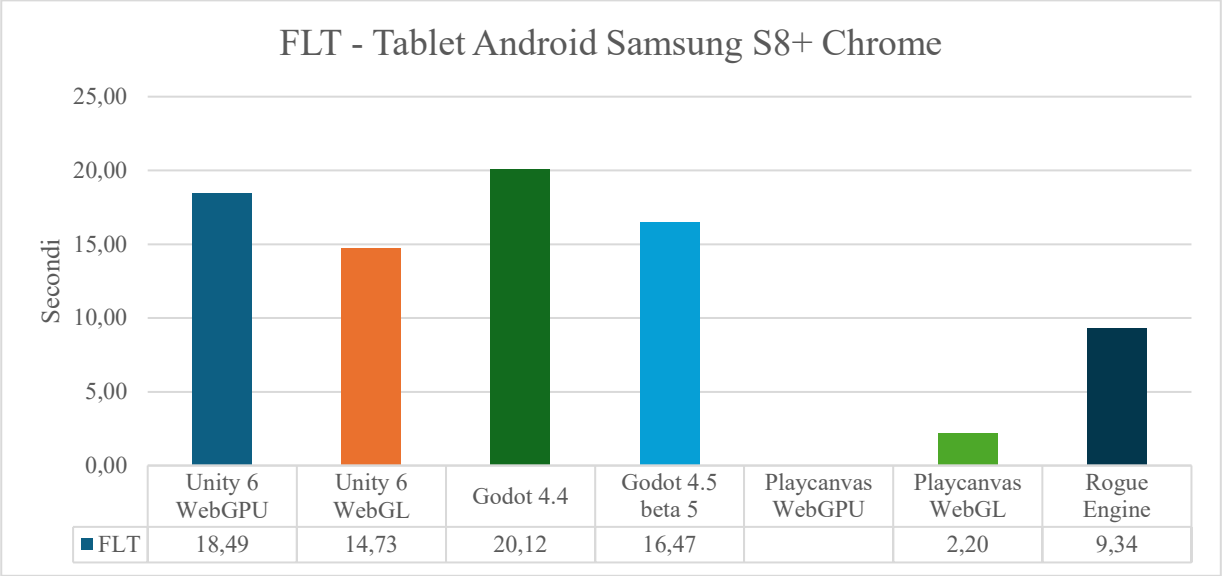



Tablet Android Samsung s8+, test effettuato con browser Chrome:

| | Unity 6 WebGPU | Unity 6 WebGL | Godot 4.4 | Godot 4.5 beta 5 | Playcanvas WebGPU | Playcanvas WebGL | Rogue Engine |
|---|---|---|---|---|---|---|---|
| media FPS | 40,27 | 54,62 | 36,80 | 37,38 | black screen | 40,02 | 65,98 |
| media frame time | 31,65 | 20,50 | 27,91 | 27,56 | | 26,35 | 30,78 |
| minimo FPS | 14 | 25 | 29 | 31 | | 17 | 2 |
| massimo FPS | 100 | 111 | 44 | 44 | | 120 | 120 |
| deviazione standard FPS | 21,36 | 20,03 | 5,77 | 6,01 | | 28,86 | 20,11 |
| FLT | 18,49 | 14,73 | 20,12 | 16,47 | | 2,20 | 9,34 |

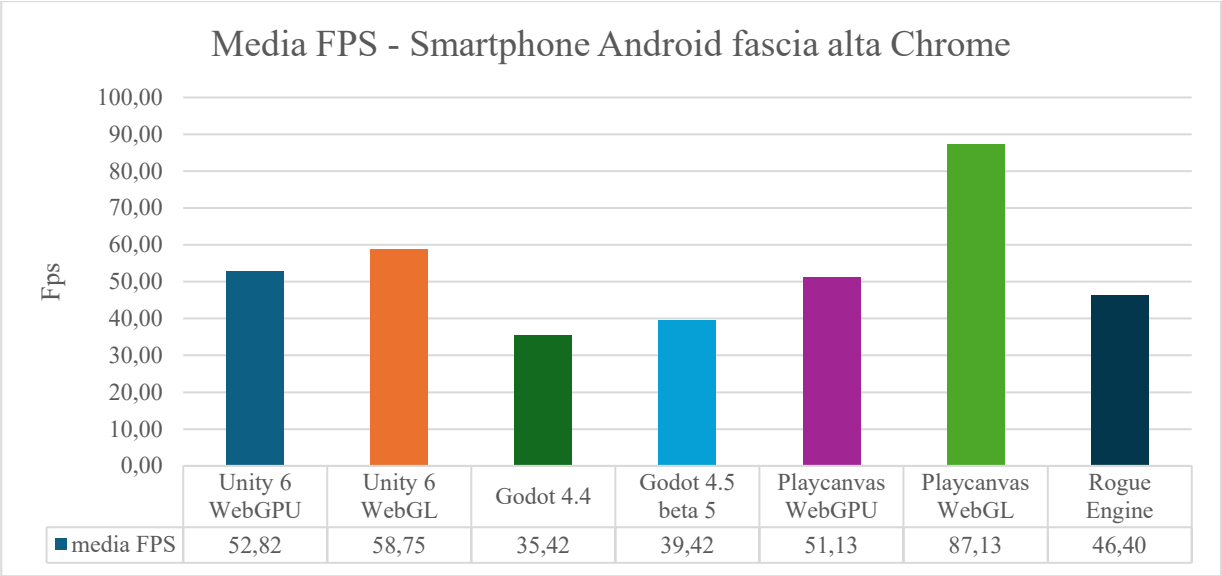


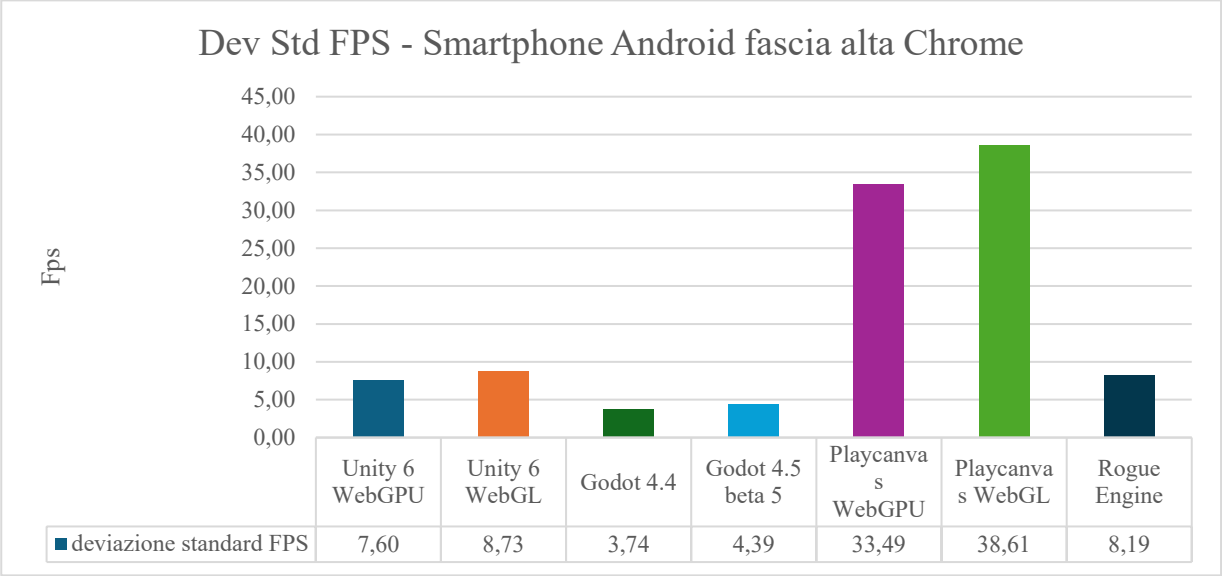


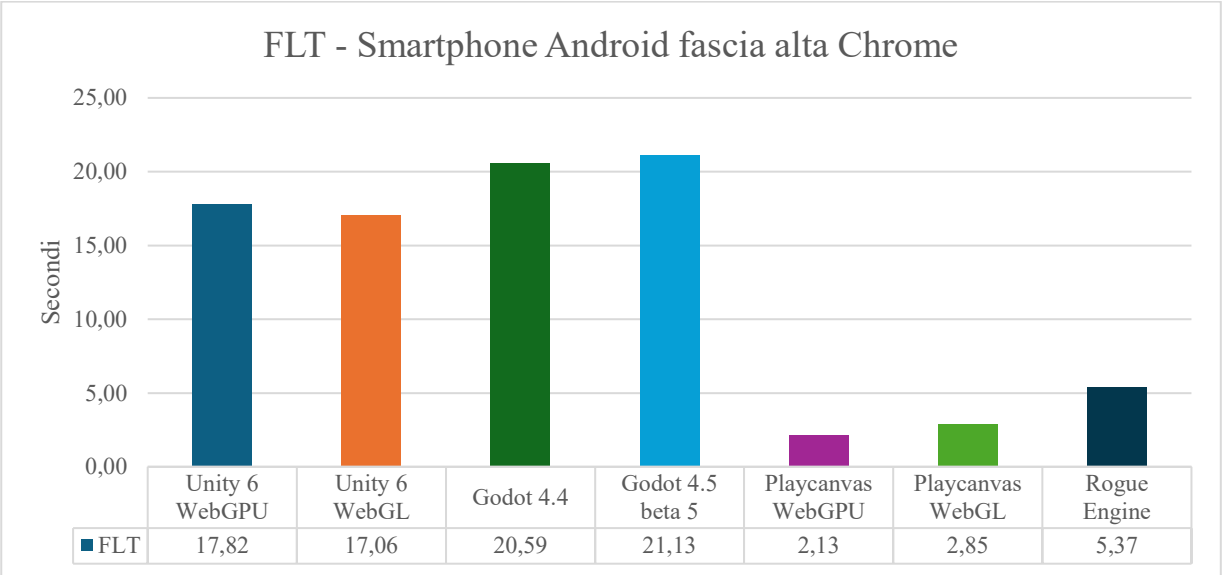




Smartphone Android fascia alta, test effettuato con browser Chrome:

| | Unity 6 WebGPU | Unity 6 WebGL | Godot 4.4 | Godot 4.5 beta 5 | Playcanvas WebGPU | Playcanvas WebGL | Rogue Engine |
|---|---|---|---|---|---|---|---|
| media FPS | 52,82 | 58,75 | 35,42 | 39,42 | 51,13 | 87,13 | 46,40 |
| media frame time | 19,42 | 17,62 | 28,65 | 25,81 | 29,95 | 16,60 | 22,59 |
| minimo FPS | 32 | 22 | 28 | 31 | 11 | 15 | 14 |
| massimo FPS | 71 | 77 | 44 | 51 | 122 | 122 | 69 |
| deviazione standard FPS | 7,60 | 8,73 | 3,74 | 4,39 | 33,49 | 38,61 | 8,19 |
| FLT | 17,82 | 17,06 | 20,59 | 21,13 | 2,13 | 2,85 | 5,37 |

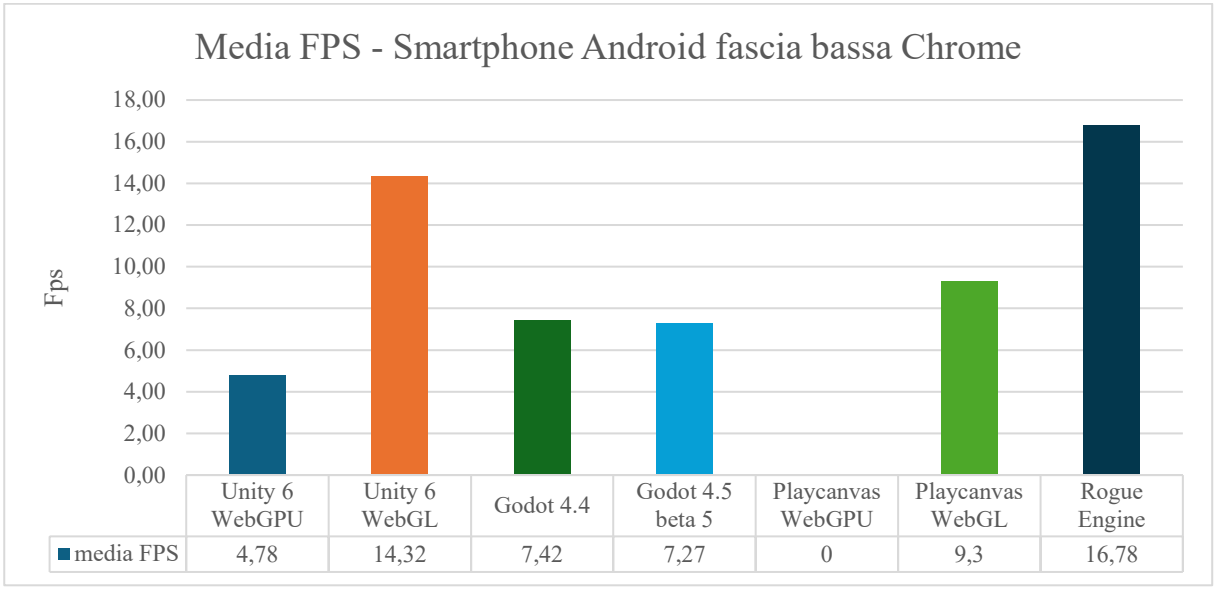

Media FPS - Smartphone Android fascia alta Chrome
Fps
100,00
90,00
80,00
70,00
60,00
50,00
40,00
30,00
20,00
10,00
0,00
Unity 6 WebGPU
Unity 6 WebGL
Godot 4.4
Godot 4.5 beta 5
Playcanvas WebGPU
Playcanvas WebGL
Rogue Engine
media FPS
52,82
58,75
35,42
39,42
51,13
87,13
46,40

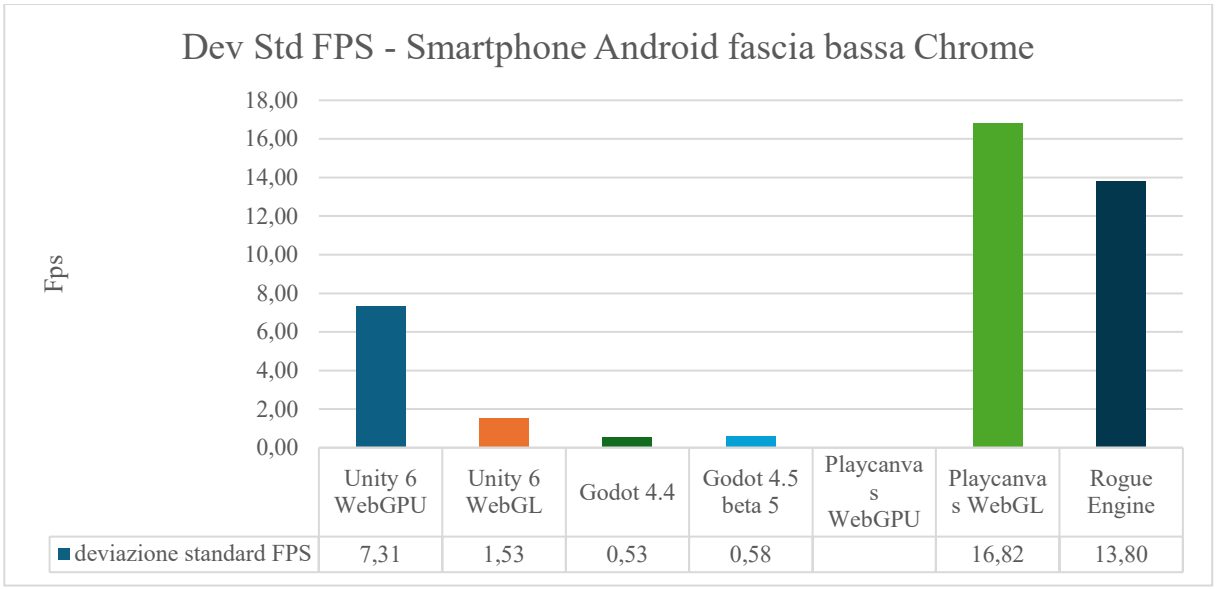

Dev Std FPS - Smartphone Android fascia alta Chrome
Fps
45,00
40,00
35,00
30,00
25,00
20,00
15,00
10,00
5,00
0,00
Unity 6 WebGPU
Unity 6 WebGL
Godot 4.4
Godot 4.5 beta 5
Playcanvas WebGPU
Playcanvas WebGL
Rogue Engine
deviazione standard FPS
7,60
8,73
3,74
4,39
33,49
38,61
8,19

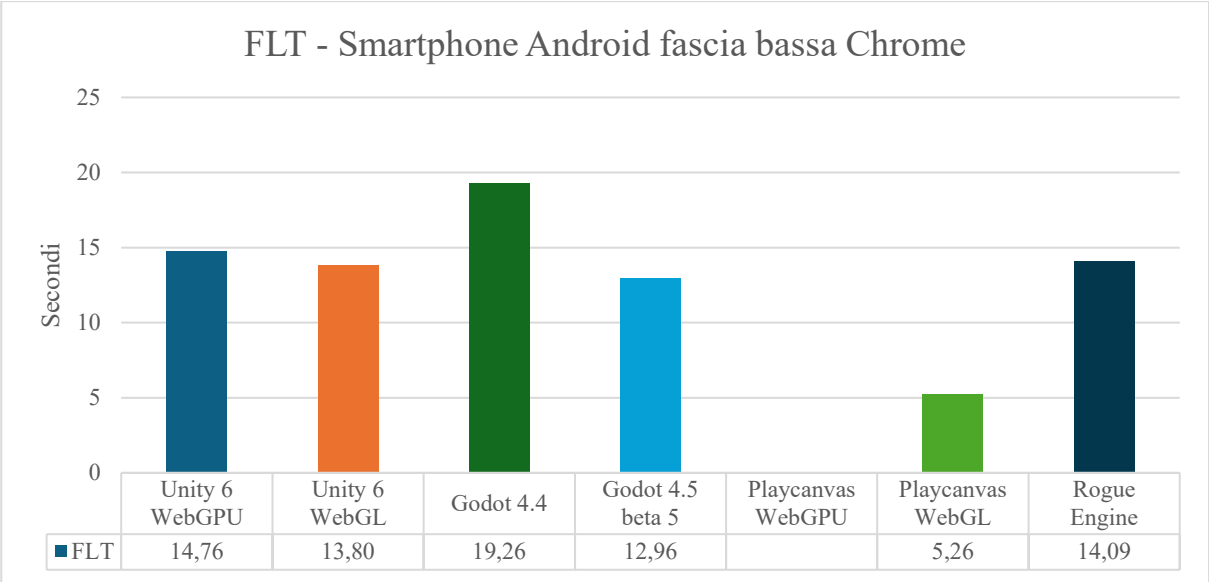

FLT - Smartphone Android fascia alta Chrome
Secondi
25,00
20,00
15,00
10,00
5,00
0,00
Unity 6 WebGPU
Unity 6 WebGL
Godot 4.4
Godot 4.5 beta 5
Playcanvas WebGPU
Playcanvas WebGL
Rogue Engine
FLT
17,82
17,06
20,59
21,13
2,13
2,85
5,37

Smartphone Android fascia bassa, test effettuato con browser Chrome:

| | Unity 6 WebGPU | Unity 6 WebGL | Godot 4.4 | Godot 4.5 beta 5 | Playcanvas WebGPU | Playcanvas WebGL | Rogue Engine |
|---|---|---|---|---|---|---|---|
| media FPS | 4,78 | 14,32 | 7,42 | 7,27 | black screen | 9,3 | 16,78 |
| media frame time | 365,60 | 70,85 | 135,72 | 138,82 | | 320,63 | 99,78 |
| minimo FPS | 2 | 9 | 7 | 5 | | 2 | 1 |
| massimo FPS | 33 | 16 | 9 | 8 | | 61 | 60 |
| deviazione standard FPS | 7,31 | 1,53 | 0,53 | 0,58 | | 16,82 | 13,80 |
| FLT | 14,76 | 13,80 | 19,26 | 12,96 | | 5,26 | 14,09 |

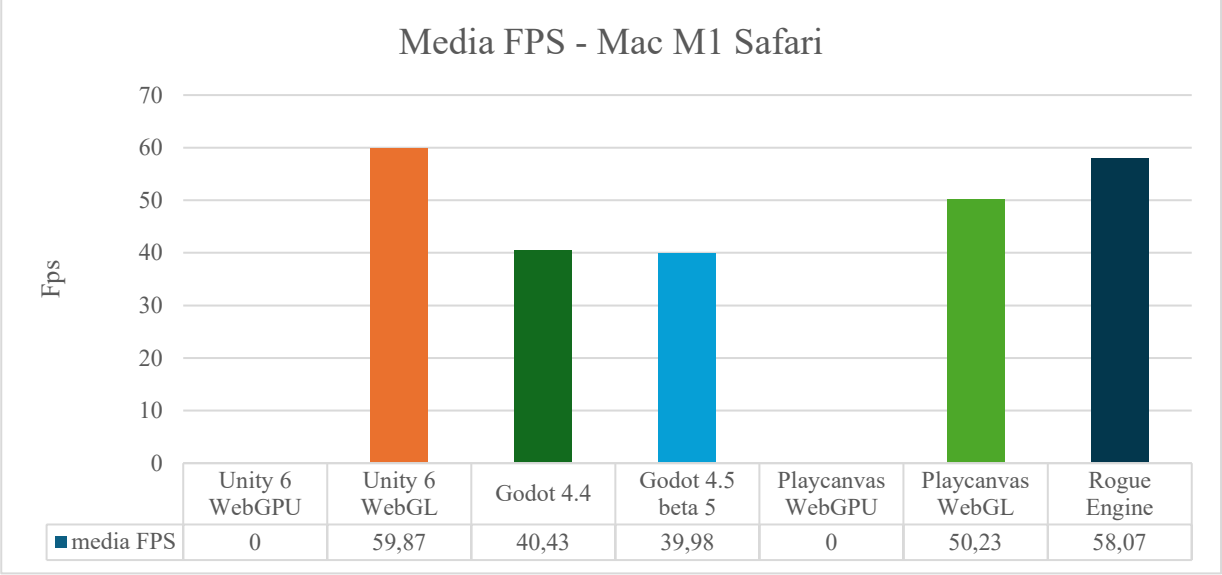


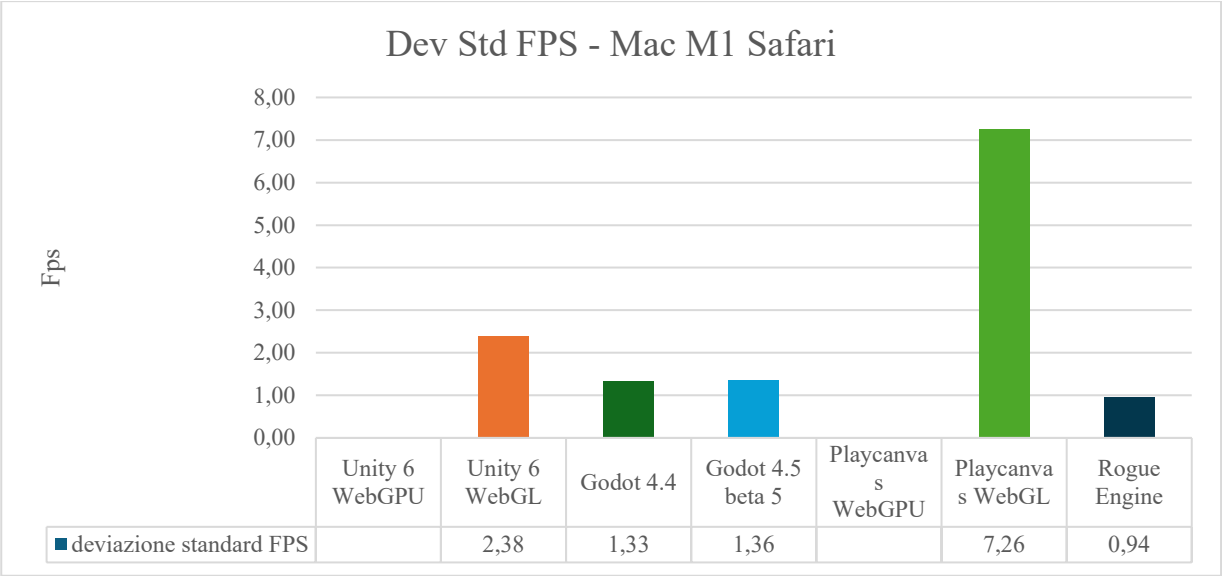



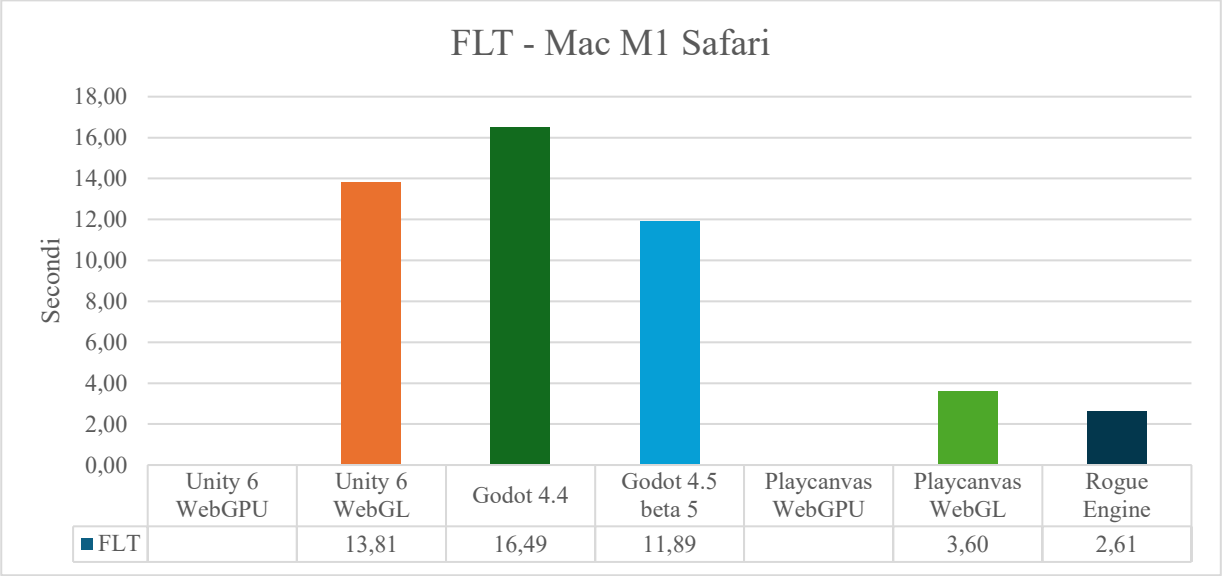



Pc portatile mac M1, test effettuato con browser Safari:

| | Unity 6 WebGPU | Unity 6 WebGL | Godot 4.4 | Godot 4.5 beta 5 | Playcanvas WebGPU | Playcanvas WebGL | Rogue Engine |
|---|---|---|---|---|---|---|---|
| media FPS | non supportato | 59,87 | 40,43 | 39,98 | non supportato | 50,23 | 58,07 |
| media frame time | | 16,73 | 24,77 | 25,05 | | 20,53 | 17,25 |
| minimo FPS | | 50 | 31 | 30 | | 29 | 56 |
| massimo FPS | | 67 | 41 | 41 | | 59 | 60 |
| deviazione standard FPS | | 2,38 | 1,33 | 1,36 | | 7,26 | 0,94 |
| FLT | | 13,81 | 16,49 | 11,89 | | 3,60 | 2,61 |

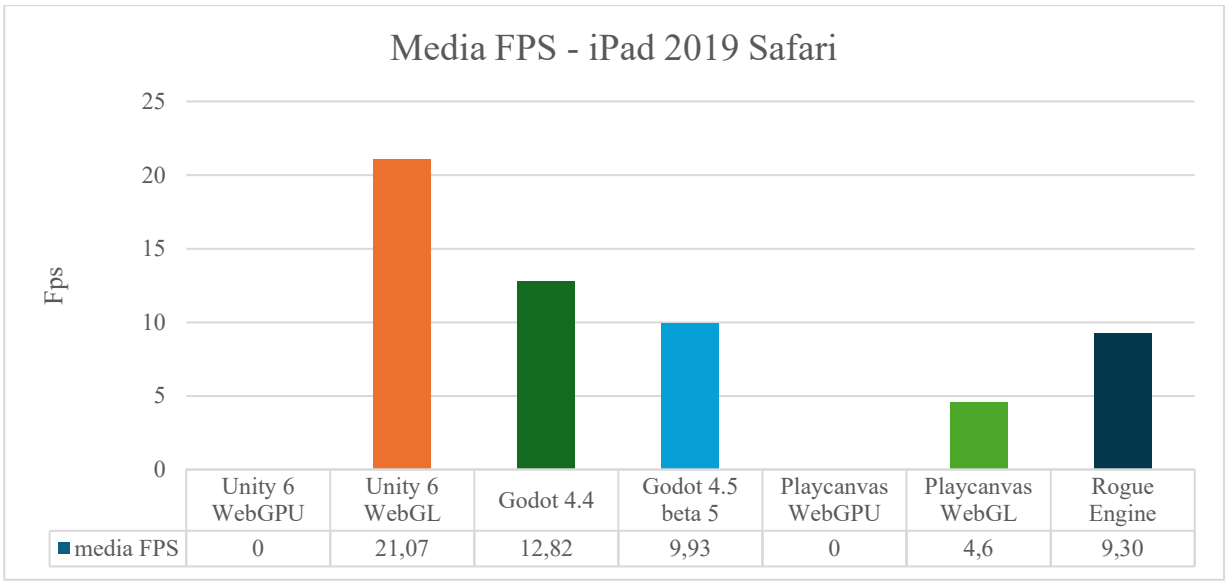


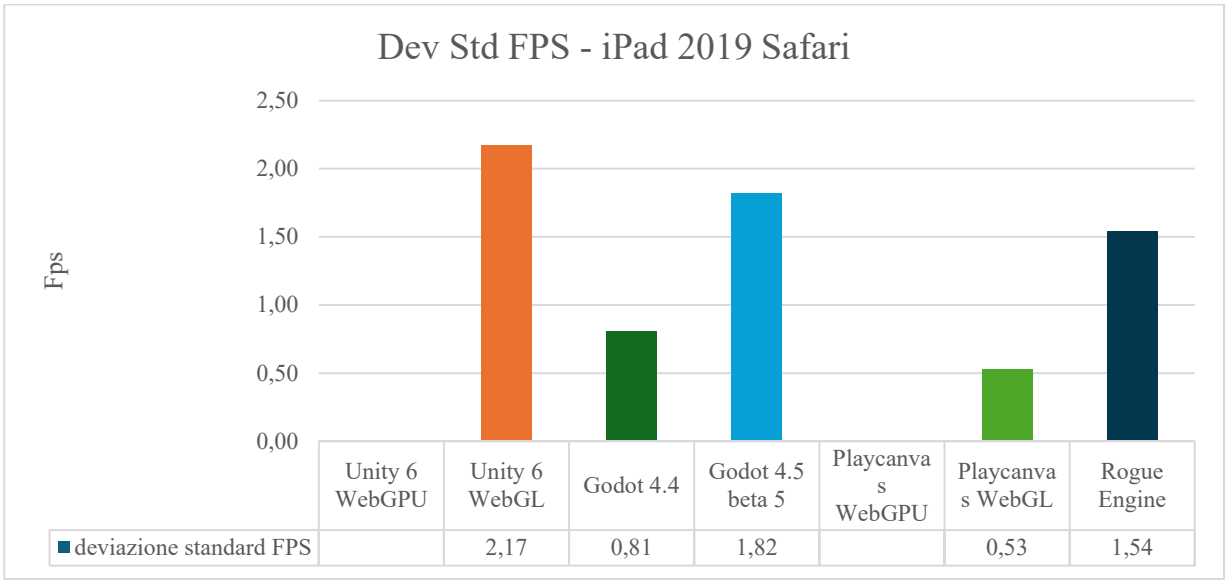


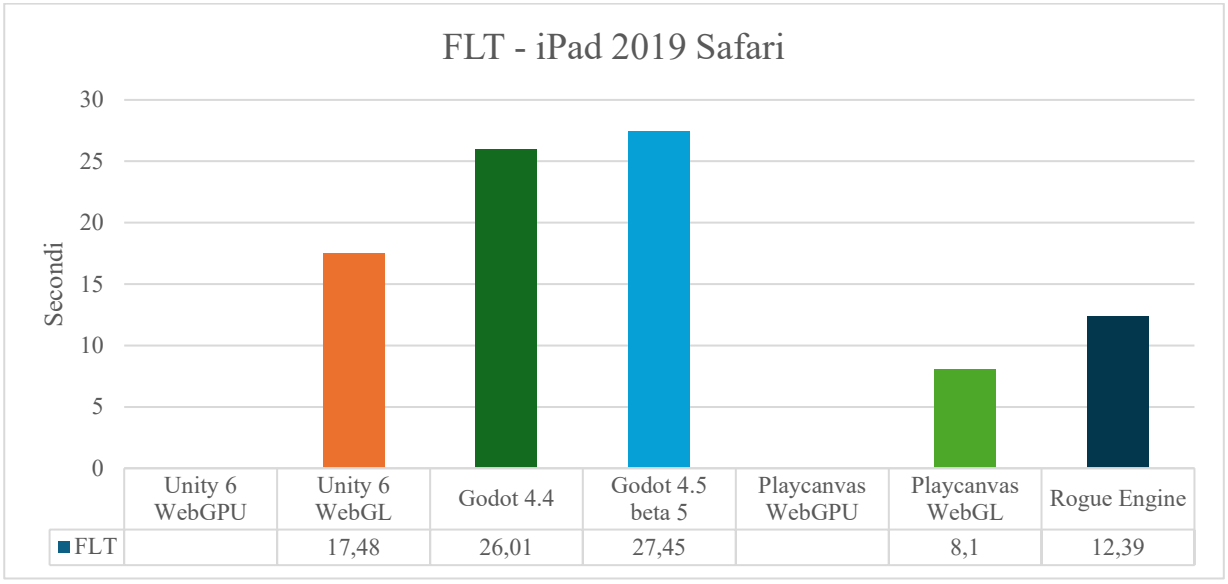




Tablet Apple iPad 2019, test effettuato con browser Safari:

| | Unity 6 WebGPU | Unity 6 WebGL | Godot 4.4 | Godot 4.5 beta 5 | Playcanvas WebGPU | Playcanvas WebGL | Rogue Engine |
|---|---|---|---|---|---|---|---|
| media FPS | non supportato | 21,07 | 12,82 | 9,93 | non supportato | 4,6 | 9,30 |
| media frame time | | 48,27 | 78,59 | 117,15 | | 217,50 | 119,77 |
| minimo FPS | | 9 | 10 | 1 | | 3 | 1 |
| massimo FPS | | 24 | 14 | 12 | | 5 | 10 |
| deviazione standard FPS | | 2,17 | 0,81 | 1,82 | | 0,53 | 1,54 |
| FLT | | 17,48 | 26,01 | 27,45 | | 8,1 | 12,39 |

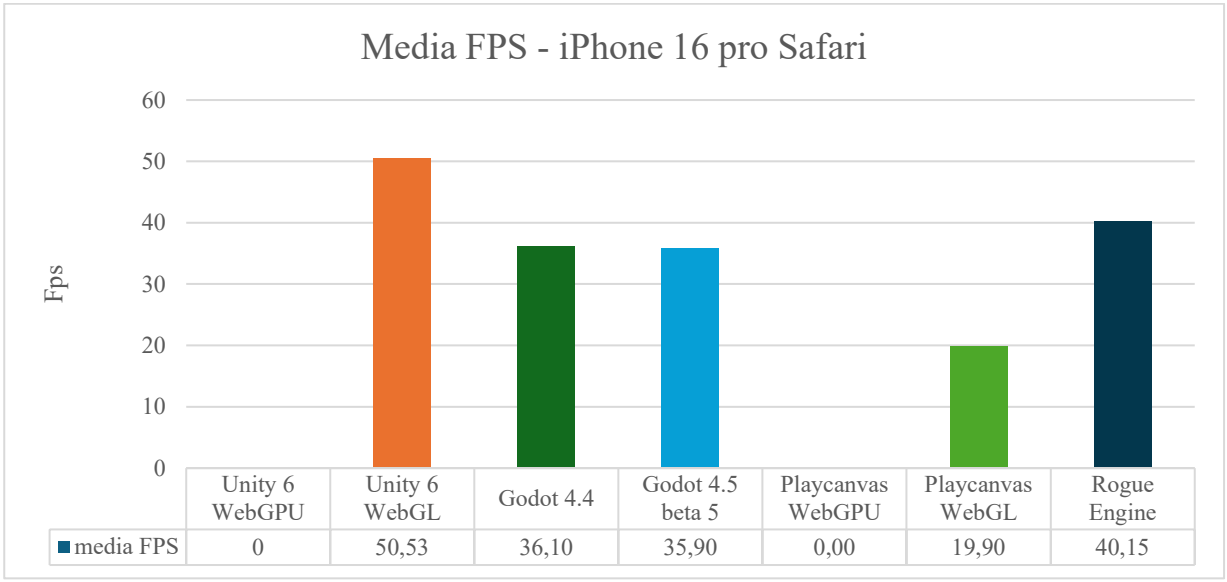

Media FPS - iPad 2019 Safari
Fps
25
20
15
10
5
0
Unity 6 WebGPU
Unity 6 WebGL
Godot 4.4
Godot 4.5 beta 5
Playcanvas WebGPU
Playcanvas WebGL
Rogue Engine
media FPS
0
21,07
12,82
9,93
0
4,6
9,30

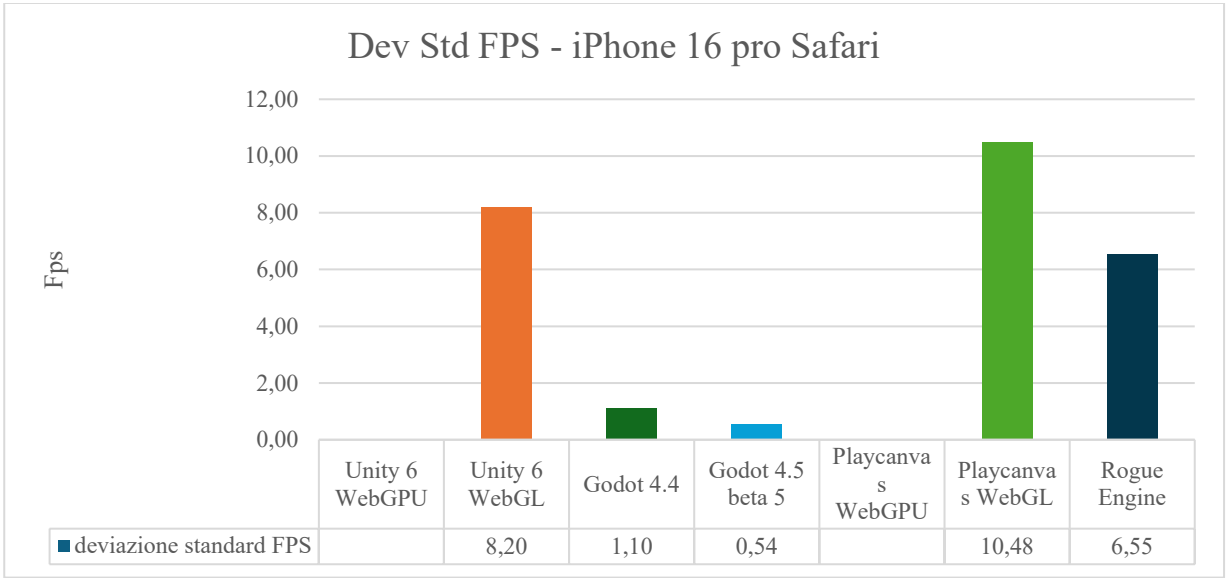

Dev Std FPS - iPad 2019 Safari
Fps
2,50
2,00
1,50
1,00
0,50
0,00
Unity 6 WebGPU
Unity 6 WebGL
Godot 4.4
Godot 4.5 beta 5
Playcanvas WebGPU
Playcanvas WebGL
Rogue Engine
deviazione standard FPS
2,17
0,81
1,82
0,53
1,54

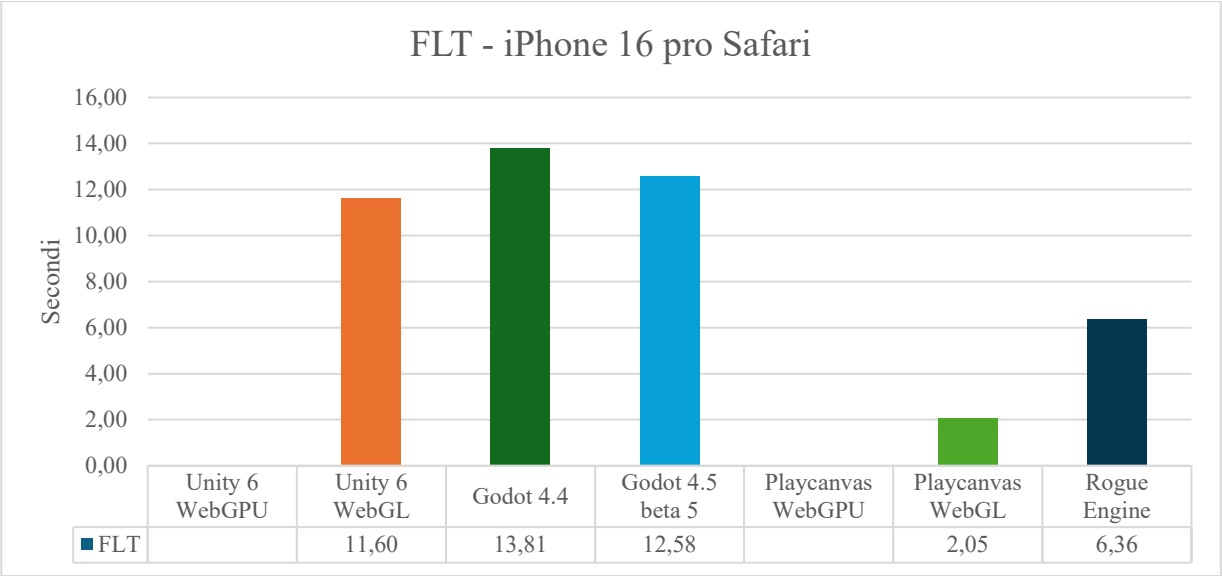

FLT - iPad 2019 Safari
Secondi
30
25
20
15
10
5
0
Unity 6 WebGPU
Unity 6 WebGL
Godot 4.4
Godot 4.5 beta 5
Playcanvas WebGPU
Playcanvas WebGL
Rogue Engine
FLT
17,48
26,01
27,45
8,1
12,39

Smartphone Apple iPhone 16 pro, test effettuato con browser Safari:

| | Unity 6 WebGPU | Unity 6 WebGL | Godot 4.4 | Godot 4.5 beta 5 | Playcanvas WebGPU | Playcanvas WebGL | Rogue Engine |
|---|---|---|---|---|---|---|---|
| media FPS | non supportato | 50,53 | 36,10 | 35,90 | non supportato | 19,90 | 40,15 |
| media frame time | | 16,25 | 8,39 | 28,16 | | 58,97 | 25,69 |
| minimo FPS | | 40,00 | 29,00 | 34,00 | | 14,00 | 26,00 |
| massimo FPS | | 63,00 | 38,00 | 37,00 | | 48,00 | 48,00 |
| deviazione standard FPS | | 8,20 | 1,10 | 0,54 | | 10,48 | 6,55 |
| FLT | | 11,60 | 13,81 | 12,58 | | 2,05 | 6,36 |



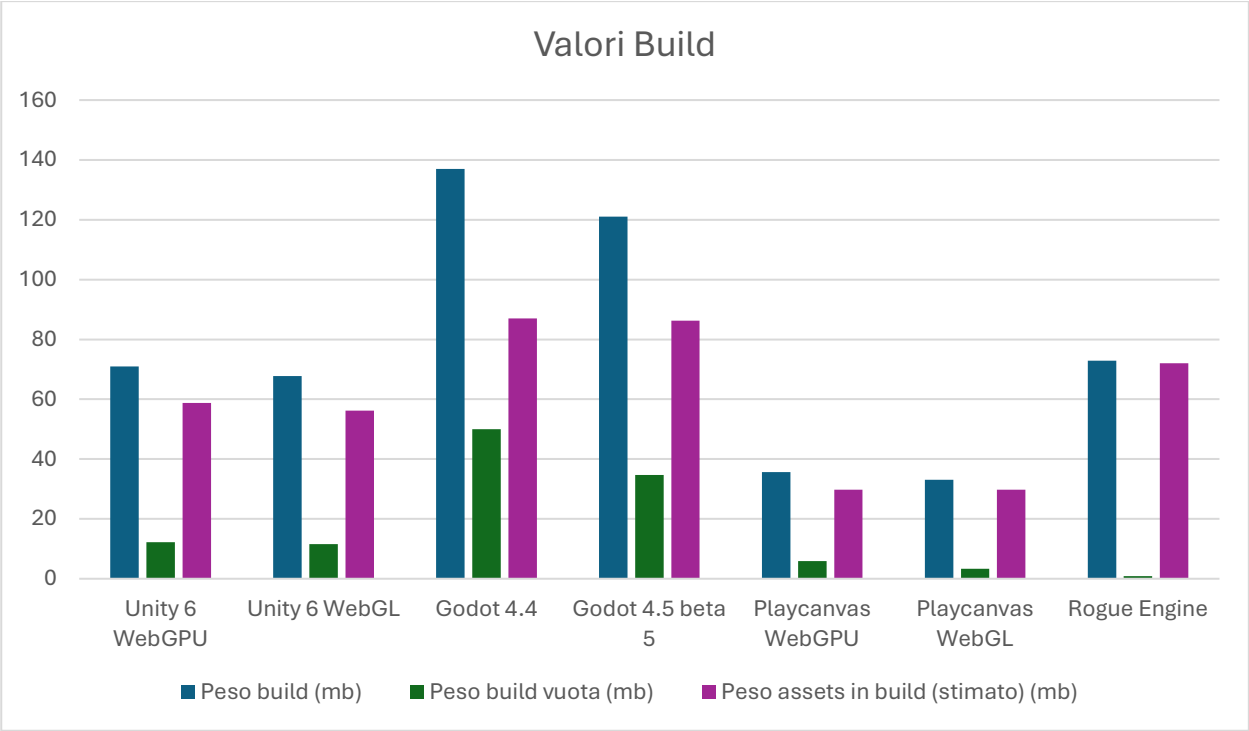






Di seguito alcuni dati generali riguardanti i singoli engine:

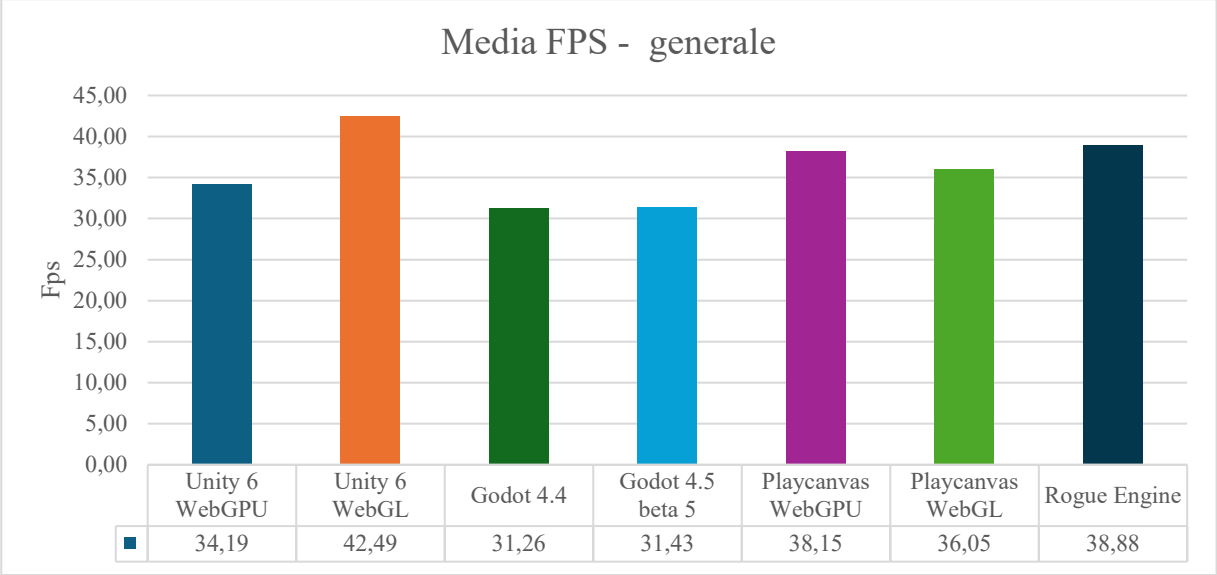
| | Unity 6 WebGPU | Unity 6 WebGL | Godot 4.4 | Godot 4.5 beta 5 | Playcanvas WebGPU | Playcanvas WebGL | Rogue Engine |
|---|---|---|---|---|---|---|---|
| Peso build (mb) | 71* | 67,8 | 137 | 121 | 35,7* | **33,1** | 72,9 |
| Peso build vuota (mb) | 12,2 | 11,6 | 50 | 34,7 | 5,92 | 3,31 | 0,826 |
| Peso assets in build (stimato) (mb) | 58,8 | 56,2 | 87 | 86,3 | 29,78 | 29,79 | 72,07 |

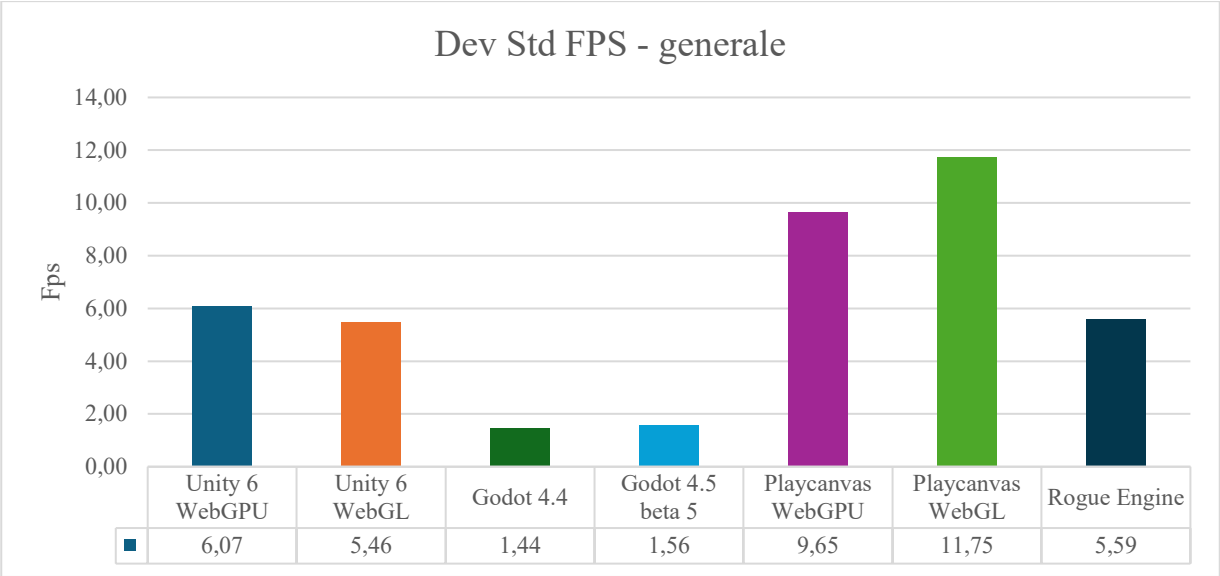
*la build comprende sia la versione WebGPU che quella WebGL

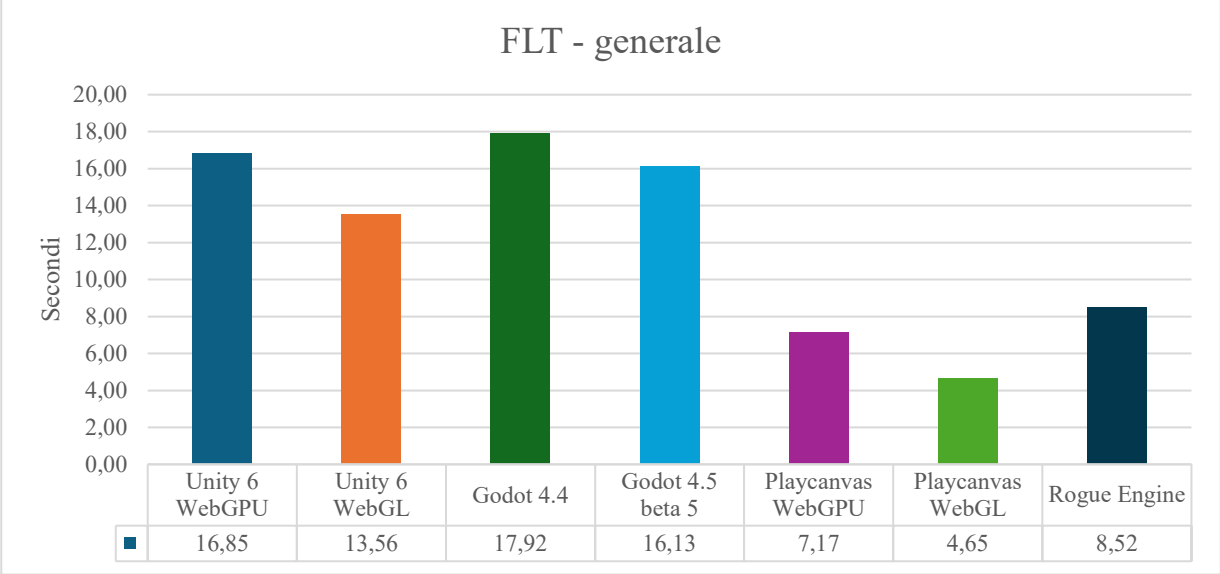



Media generale di tutti i dispositivi:

| | Unity 6 WebGPU | Unity 6 WebGL | Godot 4.4 | Godot 4.5 beta 5 | Playcanvas WebGPU | Playcanvas WebGL | Rogue Engine |
|---|---|---|---|---|---|---|---|
| media FPS | 34,19 | 42,49 | 31,26 | 31,43 | 38,15 | 36,05 | 38,88 |
| deviazione standard FPS | 6,07 | 5,46 | 1,44 | 1,56 | 9,65 | 11,75 | 5,59 |
| FLT | 16,85 | 13,56 | 17,92 | 16,13 | 7,17 | 4,65 | 8,52 |

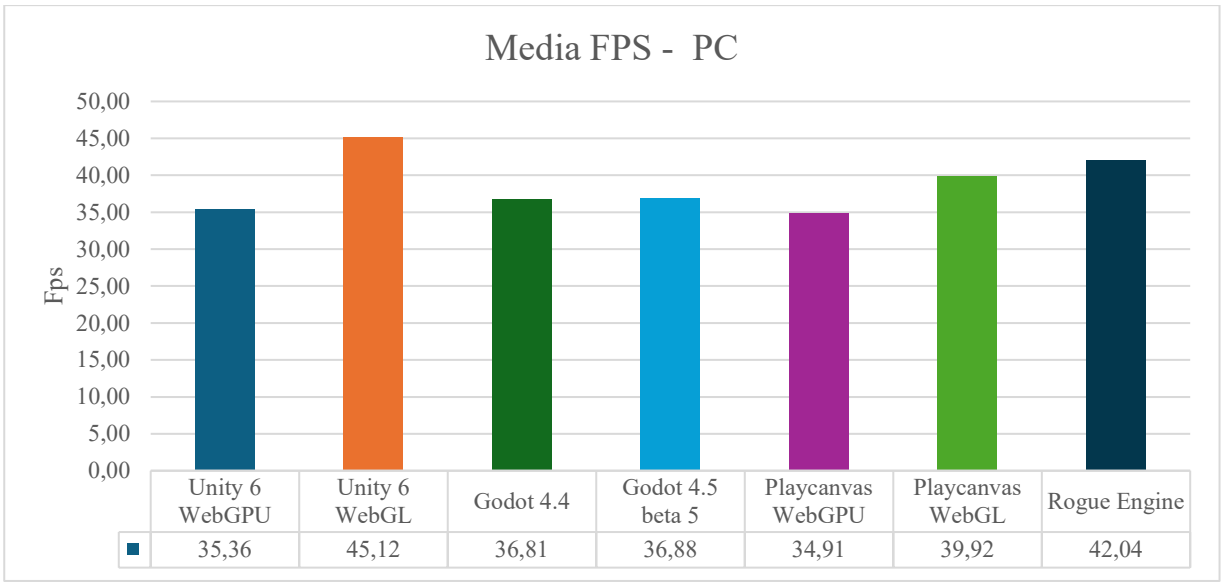


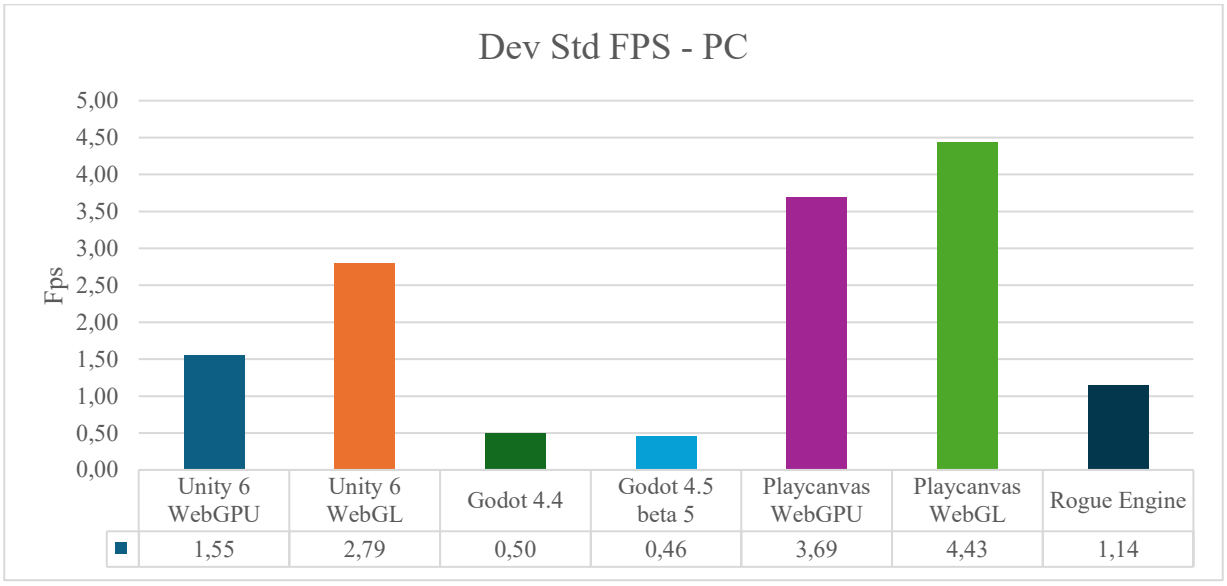


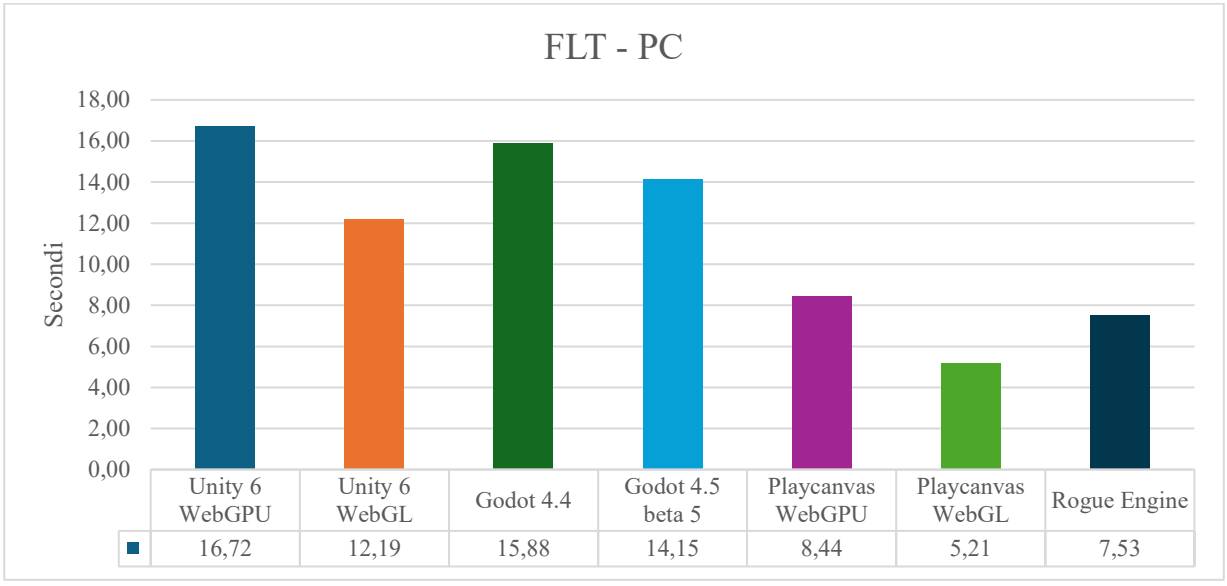




Media di tutti i computer:

| | Unity 6 WebGPU | Unity 6 WebGL | Godot 4.4 | Godot 4.5 beta 5 | Playcanvas WebGPU | Playcanvas WebGL | Rogue Engine |
|---|---|---|---|---|---|---|---|
| media FPS | 35,36 | 45,12 | 36,81 | 36,88 | 34,91 | 39,92 | 42,04 |
| deviazione standard FPS | 1,55 | 2,79 | 0,50 | 0,46 | 3,69 | 4,43 | 1,14 |
| FLT | 16,72 | 12,19 | 15,88 | 14,15 | 8,44 | 5,21 | 7,53 |

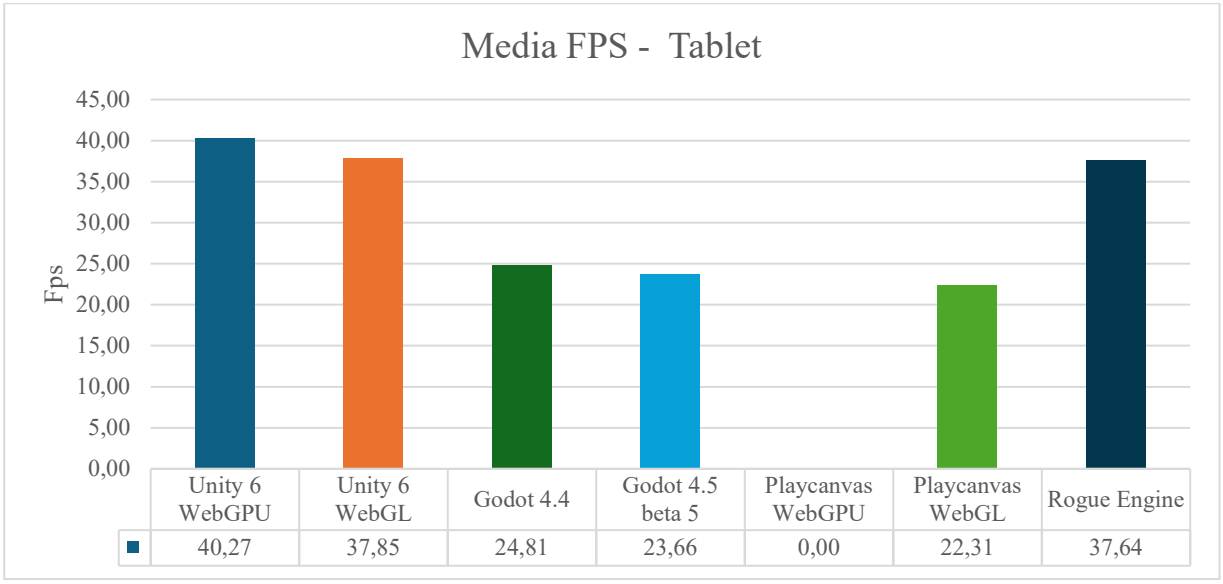


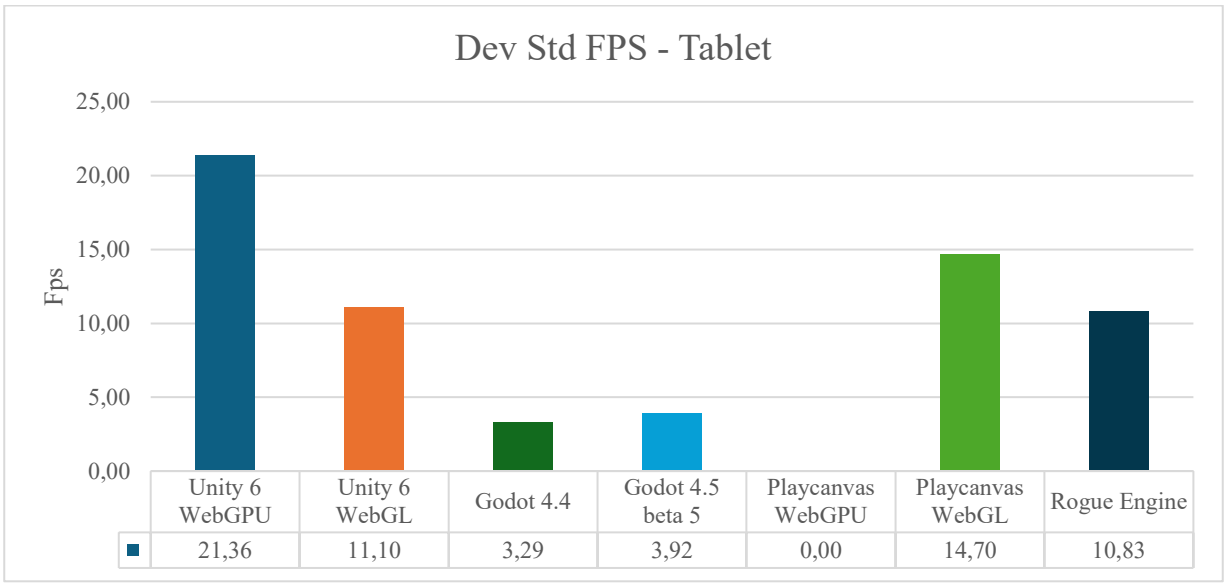



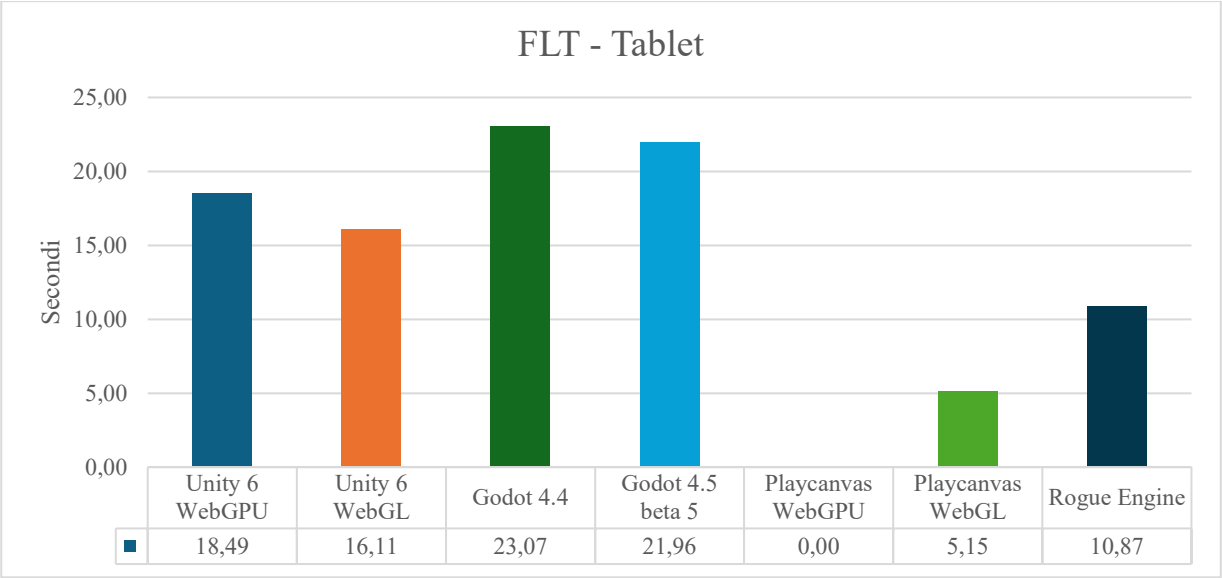




Media di tutti i tablet:

| | Unity 6 WebGPU | Unity 6 WebGL | Godot 4.4 | Godot 4.5 beta 5 | Playcanvas WebGPU | Playcanvas WebGL | Rogue Engine |
|---|---|---|---|---|---|---|---|
| media FPS | 40,27 | 37,85 | 24,81 | 23,66 | - | 22,31 | 37,64 |
| deviazione standard FPS | 21,36 | 11,10 | 3,29 | 3,92 | - | 14,70 | 10,83 |
| FLT | 18,49 | 16,11 | 23,07 | 21,96 | - | 5,15 | 10,87 |

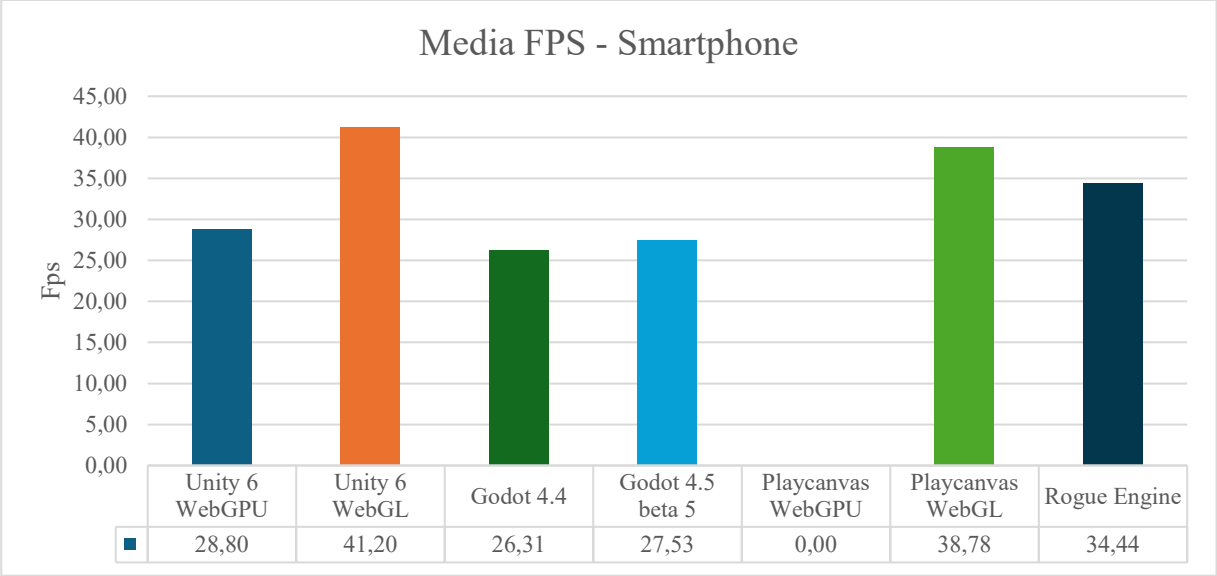


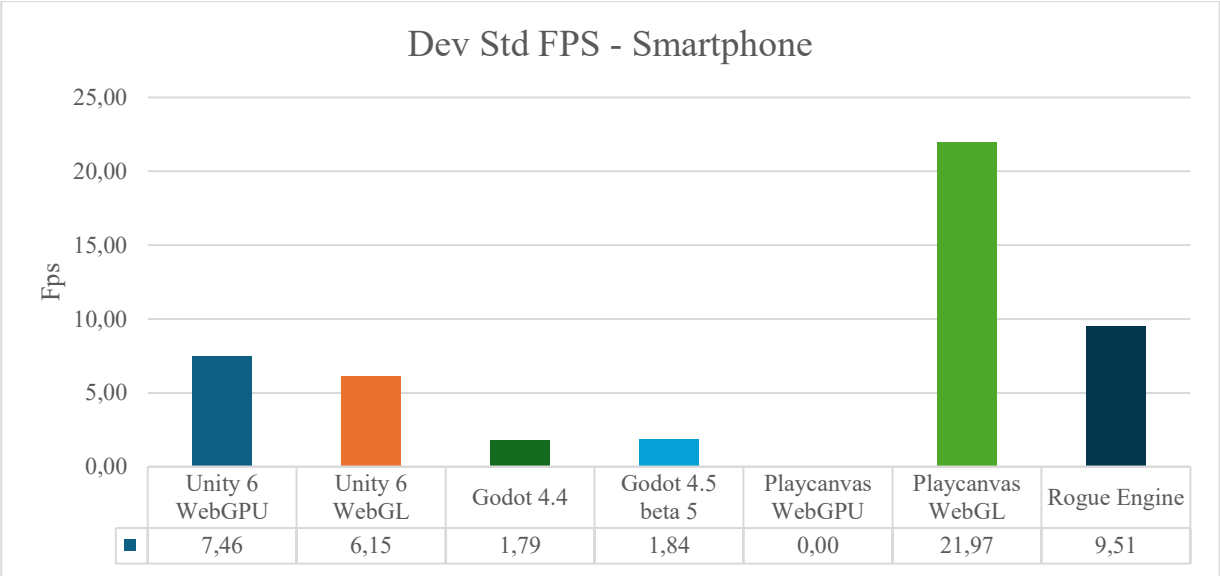



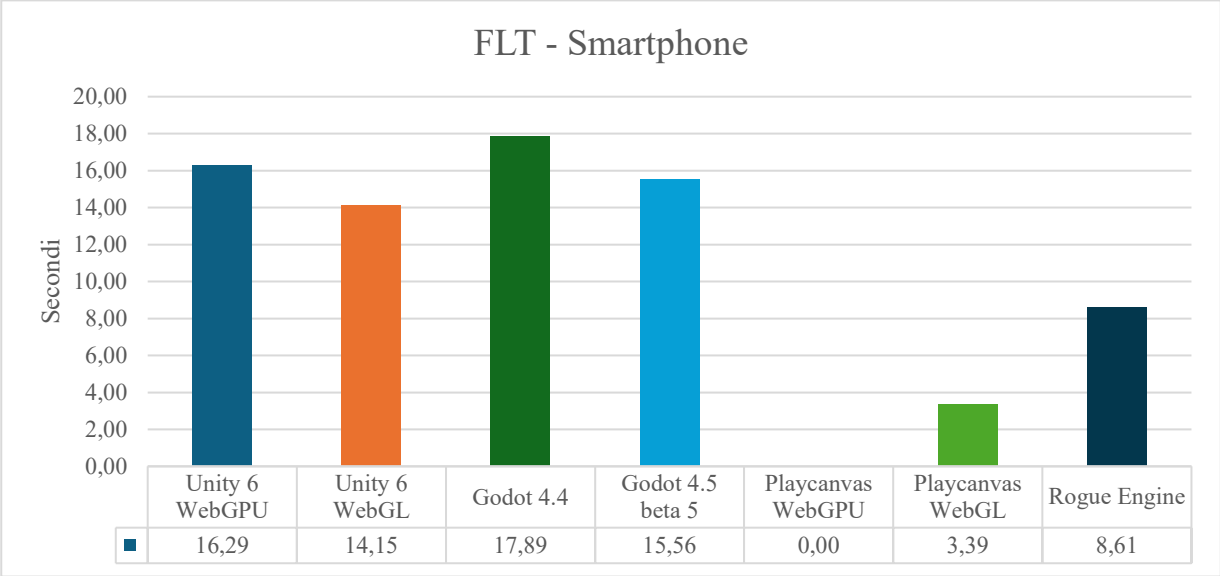



Media di tutti gli smartphone:

| | Unity 6 WebGPU | Unity 6 WebGL | Godot 4.4 | Godot 4.5 beta 5 | Playcanvas WebGPU | Playcanvas WebGL | Rogue Engine |
|---|---|---|---|---|---|---|---|
| media FPS | 28,80 | 41,20 | 26,31 | 27,53 | - | 38,78 | 34,44 |
| deviazione standard FPS | 7,46 | 6,15 | 1,79 | 1,84 | - | 21,97 | 9,51 |
| FLT | 16,29 | 14,15 | 17,89 | 15,56 | - | 3,39 | 8,61 |




Dev Std FPS - Smartphone
Fps
25,00
20,00
15,00
10,00
5,00
0,00
Unity 6 WebGPU
Unity 6 WebGL
Godot 4.4
Godot 4.5 beta 5
Playcanvas WebGPU
Playcanvas WebGL
Rogue Engine
7,46
6,15
1,79
1,84
0,00
21,97
9,51


FLT - Smartphone
Secondi
20,00
18,00
16,00
14,00
12,00
10,00
8,00
6,00
4,00
2,00
0,00
Unity 6 WebGPU
Unity 6 WebGL
Godot 4.4
Godot 4.5 beta 5
Playcanvas WebGPU
Playcanvas WebGL
Rogue Engine
16,29
14,15
17,89
15,56
0,00
3,39
8,61

# 7. Discussione dei risultati

L’analisi comparativa dei risultati sperimentali, raccolti su una vasta gamma di dispositivi e browser, consente di trarre alcune considerazioni significative riguardo alle prestazioni, alla stabilità e alla compatibilità degli engine considerati.

## 7.1 Prestazioni su PC desktop e portatili

I dati mostrano come su dispositivi desktop di fascia alta (in ambo i browser) tutti gli engine testati abbiano raggiunto valori prossimi al limite massimo di 60 FPS, con frame time medio intorno a 16–17 ms. Questo era atteso, poiché tali dispositivi dispongono di risorse hardware ampiamente sufficienti per sostenere carichi grafici moderati come quelli del prototipo.

Le differenze emergono invece osservando la deviazione standard degli FPS: Unity e Rogue Engine mostrano leggere oscillazioni (1,5–2 FPS), mentre Godot e PlayCanvas, in particolare nella versione WebGPU, si mantengono estremamente stabili, con deviazioni praticamente nulle. Ciò evidenzia come alcune implementazioni abbiano una gestione del frame pacing più uniforme.

Il discorso cambia radicalmente sui portatili di fascia media, dove gli FPS medi calano sensibilmente. In particolare, Unity WebGPU ha registrato valori estremamente bassi su Firefox (9,4 FPS), mentre Unity WebGL e PlayCanvas WebGL hanno mostrato prestazioni relativamente migliori (intorno a 15–20 FPS). Rogue Engine si è distinto con un valore medio di 16,7 FPS, posizionandosi come una soluzione più resiliente in contesti hardware limitati.

## 7.2 Prestazioni su tablet

I risultati raccolti sui tablet evidenziano una maggiore variabilità. Il caso dell’iPad 2019 è particolarmente significativo: Unity WebGL riesce a mantenere un valore medio di 21 FPS, mentre Godot e Rogue Engine scendono sotto i 13 FPS, con frame time estremamente elevati (oltre 100 ms). PlayCanvas, sorprendentemente, ha mostrato incompatibilità parziali (parlando della versione WebGPU su tablet android), limitando quindi la sua affidabilità in questo scenario.

Sul tablet Samsung S8 Plus, invece, Rogue Engine si è distinto raggiungendo 65 FPS, superando le performance di tutti gli altri engine, ma con una deviazione standard molto alta (20,11), segno di un frame rate instabile.

## 7.3 Prestazioni su smartphone

Gli smartphone hanno evidenziato la maggiore dispersione di risultati. Sul Pixel 8, Unity WebGL e PlayCanvas WebGL hanno raggiunto prestazioni vicine ai 60 FPS, mentre Godot ha mostrato un significativo calo (35–39 FPS). PlayCanvas WebGPU ha evidenziato un comportamento anomalo, con oscillazioni molto marcate (deviazione standard superiore a 30 FPS).

Su dispositivi meno performanti come l’OPPO A74, i valori scendono drasticamente: la maggior parte degli engine si attesta su 5–15 FPS, con frame time insostenibili per un’esperienza di gioco fluida. L’instabilità di PlayCanvas e Rogue Engine è particolarmente evidente su questo dispositivo.

## 7.4 Compatibilità e stabilità

Un aspetto cruciale emerso è quello della compatibilità cross-device. Gli export per il web veicolati da WebGPU non sono supportati sui dispositivi Apple con Safari riducendo la portata della loro adozione in scenari reali. C'è anche da rilevare però se che Apple si sta mobilitando per supportare questa tecnologia in versioni future del proprio software, in alcune versioni beta di Safari e iOS è possibile utilizzare WebGPU di default.

Inoltre, PlayCanvas nella sua versione WebGPU ha mostrato episodi di "black screen" su alcuni dispositivi, mentre Unity (WebGL e WebGPU) e Godot (WebGL) si sono dimostrati generalmente più stabili e supportati.

## 7.5 Tempo di caricamento (FLT)

Il First Load Time è un'altra variabile discriminante. In generale, PlayCanvas ha ottenuto i valori migliori (tra 2 e 9 secondi), grazie alla natura cloud–based e a build leggere, mentre Godot ha registrato i tempi peggiori, superando in alcuni casi i 20 secondi.

Unity si colloca in una posizione intermedia, con tempi tra i 12 e i 18 secondi, mentre Rogue Engine mostra valori contenuti (circa 8–10 secondi), dimostrando un vantaggio rispetto agli engine più strutturati.

## 7.6 Sintesi comparativa

Riassumendo, emergono alcune tendenze generali:

- Unity: prestazioni alte non sempre stabilissime, buona compatibilità anche con WebGPU, ma FLT tra i più alti;
- Godot: ottime prestazioni su PC, ma più debole e lento nei caricamenti su dispositivi mobili. Si è comunque rivelato in generale l'engine con il frame pacing migliore;
- PlayCanvas: molto rapido nei tempi di caricamento e performante in WebGL, ma con forti limiti di compatibilità e stabilità in WebGPU;
- Rogue Engine: sorprendentemente competitivo su alcuni dispositivi (es. tablet Samsung), ma penalizzato da instabilità nella fisica, nella gestione delle animazioni e nell'inaffidabilità generale dell'ambiente di sviluppo e del prodotto software finale, oltre che da un FLT solo discreto.

# 8. Conclusioni e Sviluppi Futuri

Di seguito adesso andiamo a sintetizzare il lavoro di tesi elencandone le conclusioni.

## 8.1 Risposte alle domande di ricerca

Il lavoro di tesi è stato guidato da tre domande di ricerca principali.

La prima (D1) indagava le prestazioni effettive dei principali motori di gioco quando utilizzati per il deploy di applicazioni 3D sul web. I dati raccolti hanno mostrato come tutti gli engine testati siano in grado di raggiungere i 60 FPS su dispositivi desktop di fascia alta, ma con differenze significative in termini di stabilità e tempi di caricamento. Unity si è distinto per valori medi elevati, ma con oscillazioni più marcate rispetto ad altri engine. Godot, al contrario, ha mostrato prestazioni più contenute ma con una stabilità esemplare, mentre PlayCanvas si è rivelato competitivo soprattutto nei tempi di caricamento e nella leggerezza delle build. Rogue Engine, infine, ha confermato la sua natura sperimentale, con prestazioni altalenanti e limitazioni in fase di sviluppo.

La seconda (D2) chiedeva in che misura esistano differenze tra engine nativi con supporto WebGL/WebGPU e engine nativamente orientati al web. I risultati confermano l'ipotesi formulata: gli engine nativi (Unity, Godot) offrono un ecosistema di sviluppo maturo, ma subiscono penalizzazioni quando esportati per il web, in particolare nei tempi di caricamento e nelle build più pesanti. Gli engine web-native (PlayCanvas, Rogue Engine) hanno invece dimostrato una maggiore rapidità di avvio e leggerezza delle build, ma con limitazioni legate a stabilità, compatibilità e disponibilità di funzionalità avanzate.

La terza (D3) mirava a individuare criteri per orientare la scelta di un motore di gioco in base al contesto applicativo. I dati raccolti mostrano che non esiste un "migliore" engine in senso assoluto, bensì un equilibrio tra caratteristiche. Unity rappresenta la scelta più affidabile per progetti che richiedono robustezza e completezza di strumenti; Godot è indicato in scenari in cui la stabilità e la leggerezza del codice hanno priorità, anche a scapito della performance massima; PlayCanvas risulta particolarmente utile quando il vincolo principale è la dimensione della build o la velocità di caricamento.

## 8.2 Conclusioni principali

Le ipotesi di ricerca sono state confermate. I motori nativi presentano build più pesanti e tempi di caricamento superiori, ma garantiscono un ambiente di sviluppo ricco e consolidato. Gli engine web-native, viceversa, si distinguono per tempi di avvio più rapidi e maggiore integrazione con le tecnologie web, pur con limiti legati a stabilità e feature. Nel confronto diretto, Unity è risultato l'engine con le migliori prestazioni complessive, pur mostrando una maggiore instabilità del framerate. Godot si è collocato sul versante opposto, con valori più bassi ma estremamente stabili. PlayCanvas ha confermato la sua natura di soluzione snella e veloce, caratterizzata da build ridotte e FLT contenuti, mentre Rogue Engine si è rivelato troppo immaturo per progetti complessi, sia dal punto di vista dello sviluppo che del deploy.

Dal punto di vista pratico e industriale, i risultati si rivelano preziosi per supportare decisioni strategiche. In un'azienda come Tiny Bull Studios, dove la produzione di giochi web è costante, mantenere Unity come piattaforma di riferimento appare una scelta ragionevole, soprattutto in termini di affidabilità. Allo stesso tempo, l'utilizzo di PlayCanvas può essere vantaggioso per progetti in cui la dimensione della build e i tempi di caricamento rappresentano fattori critici.

## 8.3 Limiti dello studio

Come ogni ricerca sperimentale, anche questo lavoro presenta limiti che devono essere considerati. Sul piano metodologico, i test si sono concentrati su un solo prototipo 3D (Knight Runner), lasciando aperta la questione della generalizzabilità dei risultati a generi differenti, come i giochi 2D. Inoltre, le sessioni di test sono state di durata relativamente breve (tre minuti), e questo potrebbe non riflettere comportamenti a lungo termine, ad esempio in termini di gestione della memoria o stabilità di rete. Un ulteriore limite riguarda le metriche raccolte: mentre gli FPS, il frame time e il FLT sono stati monitorati con precisione, non è stato possibile misurare il consumo di memoria e CPU, che avrebbero richiesto strumenti esterni più complessi. Questi dati, se disponibili, avrebbero arricchito l'analisi fornendo un quadro più completo delle prestazioni. Infine, alcune versioni degli engine considerate (ad esempio Godot 4.5 beta 5) erano in fase non definitiva al momento della sperimentazione. Ciò introduce un margine di incertezza, poiché le ottimizzazioni future potrebbero alterare le prestazioni osservate.

## 8.4 Lavori futuri e raccomandazioni

Gli sviluppi futuri di questa ricerca possono muoversi in diverse direzioni. In primo luogo, sarebbe utile estendere lo studio includendo prototipi 2D, così da verificare se le differenze osservate nei motori si confermano anche in scenari di complessità inferiore. In secondo luogo, l'integrazione di metriche aggiuntive, come consumo di memoria, utilizzo della GPU e impatto energetico, permetterebbe di ottenere un quadro più ampio e vicino alle esigenze reali degli sviluppatori.

Dal punto di vista tecnologico, un'evoluzione interessante sarà rappresentata dalla diffusione di WebGPU come standard stabile. Ripetere i test quando WebGPU sarà ampiamente supportato consentirà di comprendere se questo cambiamento potrà ridurre il gap prestazionale tra motori nativi e web-native. Allo stesso modo, includere engine non testati in questo lavoro (ad esempio Babylon.js) permetterebbe di avere un panorama ancora più rappresentativo.

Le aziende interessate allo sviluppo di giochi web dovrebbero considerare attentamente il bilanciamento tra prestazioni, tempi di caricamento e complessità degli strumenti di sviluppo: Unity rimane una scelta sicura e consolidata, PlayCanvas si rivela strategico per progetti leggeri e distribuiti rapidamente, mentre Rogue Engine richiede ancora maturazione prima di un impiego su larga scala.

# 9. Bibliografia


Barczak, Andrzej Marian, e Hubert Woźniak. «Comparative Study on Game Engines». *Studia Informatica. Systems and Information Technology. Systemy i Technologie Informacyjne*, fasc. 1–2 (2019). https://doi.org/10.34739/si.2019.23.01.

Berdak, Przemysław, e Małgorzata Plechawska-Wójcik. «Performance Analysis of Unity3D Engine in the Context of Applications Run in Web Browsers». *Journal of Computer Sciences Institute* 5 (dicembre 2017): 167–73. https://doi.org/10.35784/jcsi.616.

Bi, Weichen, Yun Ma, Deyu Tian, Qi Yang, Mingtao Zhang, e Xiang Jing. «Demystifying Mobile Extended Reality in Web Browsers: How Far Can We Go?» *Proceedings of the ACM Web Conference 2023*, ACM, 30 aprile 2023, 2960–69. https://doi.org/10.1145/3543507.3583329.

Engine, Godot. «Upcoming (Serious) Web Performance Boost». Godot Engine. Consultato 17 settembre 2025. https://godotengine.org/article/upcoming-serious-web-performance-boost/.

Halsas, Rasmus. *Comparison of HTML5 Game Engines Used in Game Development*. 2017. https://lutpub.lut.fi/handle/10024/130917.

Jangda, Abhinav, Bobby Powers, Emery Berger, e Arjun Guha. «Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code». Preprint, 31 maggio 2019. https://doi.org/10.5555/3358807.3358817.

Johansson, Julia. *Performance and Ease of Use in 3D on the Web : Comparing Babylon.Js with Three.Js*. 2021. https://urn.kb.se/resolve?urn=urn:nbn:se:bth-20977.

Kim, Woo Jae, e Bohdan B. Khomtchouk. «WebAssembly enables low latency interoperable augmented and virtual reality software». arXiv:2110.07128. Preprint, arXiv, 2 dicembre 2024. https://doi.org/10.48550/arXiv.2110.07128.

Mehanna, Naif, e Walter Rudametkin. «Caught in the Game: On the History and Evolution of Web Browser Gaming». *Companion Proceedings of the ACM Web Conference 2023* (New York, NY, USA), WWW '23 Companion, Association for Computing Machinery, 30 aprile 2023, 601–9. https://doi.org/10.1145/3543873.3585572.

Mohd, Tauheed Khan, Fernando Bravo-Garcia, Landen Love, Mansi Gujadhur, e Jason Nyadu. «Analyzing Strengths and Weaknesses of Modern Game Engines». *International Journal of Computer Theory and Engineering* 15, fasc. 1 (2023): 54–60. https://doi.org/10.7763/IJCTE.2023.V15.1330.

Pattrasitidecha, Akekarat. *Comparison and Evaluation of 3D Mobile Game Engines*. 2014. https://hdl.handle.net/20.500.12380/193979.

Tufegdžić, Janko, Matija Dodović, Mihajlo Ogrizović, Nikola Babić, Jovan Đukić, e Dražen Drašković. «Application of WebAssembly Technology in High-Performance Web Applications». *2024 11th International Conference on Electrical, Electronic and Computing Engineering (IcETRAN)*, giugno 2024, 1–6. https://doi.org/10.1109/IcETRAN62308.2024.10645198.

«Unreal Engine 4.23 released!» Consultato 21 agosto 2025. https://www2.unrealengine.com/fr/blog/unreal-engine-4-23-released.

«WebGPU». Consultato 21 agosto 2025. https://www.w3.org/TR/webgpu/.

«WebXR Device API». Consultato 21 agosto 2025. https://www.w3.org/TR/webxr/.

# 10. Immaginografia

# 11. Ringraziamenti

Per cominciare vorrei ringraziare il mio relatore, il professore Marco Mazzaglia, e l'azienda Tiny Bull Studios per avermi permesso di scrivere questa tesi e avermi seguito in questo percorso. Le sue lezioni professore hanno permesso sia di farmi crescere professionalmente sia di far crescere ulteriormente la mia passione per lo sviluppo di videogame.

Ringrazio la mia famiglia: Gaia, mamma e papà, nonna Filomena, nonna Maria e nonno Antonio, zia Michela, zio Nicola, Ilario e Francesca. Grazie per avermi permesso di intraprendere questo percorso di laurea magistrale a Torino, grazie per avermi supportato sempre in questi due anni, grazie per tutti i voli presi a orari improponibili per andare e tornare così da non sentire mai la mancanza di casa.

Grazie a tutti i miei colleghi che mi hanno accompagnato in questi due anni, in particolare grazie a Chiara, Giuseppe, Andrea, Francesco B., Gabriele, Sara e Tommaso; grazie per avermi accolto fin da subito e per tutti gli esami condivisi assieme. Ringrazio anche il Level Up Lab, team studentesco, che mi ha dato la spinta fin da subito a coltivare la mia passione per lo sviluppo videogame. Grazie al team dei miei sogni: 2Hardware, vi ringrazio per tutto il tempo passato insieme e per il gioco che siamo riusciti a portare a termine, grazie per avermi fatto credere e vivere il mio sogno. Un grazie particolare a Simone, Roberto e Francesco C.; grazie per essermi stati vicini anche nei momenti informali, grazie per le serate passate insieme e per avermi supportato e sopportato in questi 2 anni.

Grazie Cristian, Alessandra e Matteo. Ho iniziato con voi questa avventura a Torino e non potevo chiedere compagnia migliore, grazie per tutte le sere a casa vostra e per tutte le pizze di Candido condivise assieme.

Ringrazio tutta la Gi.Fra. di Torino, in particolare Mattia, Ludovica, Roberta R., Agnese, Alessia, Francesca, Claudio, Ivana, Giulia, Sofia. Grazie per avermi accolto e avermi fatto respirare aria francescana anche molto lontano da casa, siete stati la mia ancora nei momenti di nostalgia. Ringrazio anche tutti i gifrini sparsi nel mondo che mi hanno supportato in questi anni, grazie Marco, Domenico e Chiara.

Grazie Roberta, il mio primo collegamento con la città di Torino, avvenuto in tempi non sospetti; grazie per avermi accolto e dedicato sempre del tempo, grazie per tutti i passaggi in macchina e per tutto quello che hai fatto per me in questi mesi passati assieme.

Grazie anche a tutti i frati e le suore che mi sono stati accanto in questi mesi e che hanno pregato per me, grazie Fra Daniele, Fra Dario, Suor Chiara, Suor Lucia, tutte le suore angeline di Torino, Fra Sabino.

Ringrazio la Gi.Fra. di Barletta, grazie per non avermi mai fatto sentire lontano, grazie per avermi accolto ogni volta che tornavo come se non fossi mai andato via, grazie per tutti i messaggi scambiati e l'amore condiviso anche a distanza.

Grazie ai miei amici più stretti, Arcangela, Nives, Paola, Rossana, Valeria, Alessandro, Ludovico, Riccardo, Dajana; grazie per esserci sempre. Grazie Gaetano per gli ultimi mesi passati assieme a Torino, è stato molto bello passarli con te e crescere insieme. Grazie Giuseppe, sei stato molto importante in tutti i momenti di gioia, nostalgia e preoccupazione; grazie per le videochiamate infinite e per tutti i pomeriggi passati a giocare insieme anche a distanza. Mi hai fatto sentire casa sempre meno distante.

Ringrazio il Signore per avermi dato la vita, la possibilità di studiare e vivere spensierato questi anni di studio. Spero di poter fare tesoro di tutto quello che ho imparato in questi anni e metterli a disposizione del prossimo per fare sempre del bene.