**01.INTRODUZIONE**

* **Cos’è Vulkan?**

E’ un API cross-platform sviluppata dal Khronos Group progettata per essere usata in contesti ad alte prestazioni sfruttando il multi-threading. L’API è supportata anche da dispositivi mobile.

* **Panoramica del tutorial**

Il codice è scritto in C++, ma non vengono usate funzionalità avanzate specifiche del linguaggio, perciò è possibile utilizzare questa guida anche in C o Rust.

* **Quando usare Vulkan?**

Se il tuo collo di bottiglia è la GPU, Vulkan non farà miracoli nel migliorare le prestazioni. Tuttavia, se bisogna renderizzare scene CAD, grandi mappe o mondi dinamici allora le capacità di parallelismo dell’API porteranno benefici.

I driver mobile per telefoni sono frammentati, solo i dispositivi più recenti implementano Vulkan e potrebbero non fornire la miglior implementazione dell’API a disposizione confrontata con il mondo Desktop. I telefoni con driver buoni beneficiano delle prestazioni di Vulkan, risparmiando anche il consumo di batteria.

* **Validation Layers**

Sono una funzionalità opzionale di vulkan, permettono di verificare se si sta usando correttamente l’API e aiutano ad individuare se ci sono errori, ad esempio uso di enumerativi errati, problemi di sincronizzazioni, controllo dell’emivita degli oggetti, configurazioni incorrette e cosi via.

Nota che i layers non inviduano errori di programmazione come memoria non inizializzata o puntatori malgestiti. E’ consigliato usare strumenti come **Address Sanitizer** (incluso in LLVM) e **Valgrind** (Linux). Tuttavia questi strumenti possono creare falsi positivi e rendere l’output rumoroso.

* **Utilizzo e mentalità**

L’API ha un design a oggetti che puoi creare e utilizzare manualmente, non solo per le risorse della GPU come **Immagini/Textures** e **Buffer** (spazi di memoria) ma anche molte **configurazioni interne.**

Ad esempio, alcune funzionalità fisse della GPU (**fixed functions**) come la modalità di rasterizzazione, sono immagazzinate in oggetti **Pipeline** che mantengono shaders e altre configurazioni. In OpenGL e DirectX 11 queste sono calcolate a runtime durante il rendering.

In Vulkan, quando scrivi questi oggetti devi pensare se metterli in cache (memoria ad accesso rapido per guadagnare prestazioni) oppure se crearli durante il rendering. Alcuni oggetti come quelli della pipeline sono dispendiosi da creare a runtime, quindi è consigliato crearli durante schermate di caricamento o thread in background. Altri oggetti sono più leggeri e possono essere creati a runtime come i **DescriptorSets**.

Siccome in Vulkan tutto è “pre-compilato”, lo stato di convalida della GPU è fatto nel momento in cui l’oggetto viene creato, e il rendering stesso farà meno lavoro e sarà più veloce.

Quando eseguiamo comandi sulla GPU, questi devono essere inseriti in un **CommandBuffer**, uno spazio di memoria dedicato ai comandi da inviare alla GPU, e essere messi in una coda (**Queue)**. Prima si alloca memoria al command buffer e si codificano le cose all’interno di esso, poi esegui il contenuto aggiungendoli alla queue.

Quando aggiungi un comando alla queue questo verrà eseguito sulla GPU e hai strumenti per controllare quando l’esecuzione è terminata. Se invii più commandi su più queue questi verranno eseguiti in parallelo.

Non c’è il concetto di **frame** in Vulkan, il modo in cui renderizzi dipende da te. L’unica cosa che conta è quando devi mostrare il fotogramma sullo schermo, tramite una **swapchain.**

Ciò significa che puoi usare vulkan anche per non renderizzare necessariamente in una finestra, ma puoi creare un rendering e salvarlo in un file per scopo di testing, o usare Vulkan a scopo computazionale come **ray tracing** o **compute shaders.**