ALMA MATER STUDIORUM - UNIVERSITÁ DI BOLOGNA

CAMPUS DI CESENA
SCUOLA DI SCIENZE

CORSO DI LAUREA IN INGEGNERIA E SCIENZE INFORMATICHE

SVILUPPO DI UN MOTORE DI RENDERING BASATO SULLA LIBRERIA VULKAN

Tesi di laurea in:
Computer Graphics

Relatrice: Presentata da:

Dott.ssa ***Damiana Lazzaro*** ***Luca Succi***

Sessione II
Anno Accademico 2017/2018

Introduzione

Questa tesi ha come obiettivo sviluppare un motore di rendering utilizzando l'API Vulkan. Vulkan è una libreria grafica di basso livello e a basso impatto, indipendente da e compatibile con tutti i sistemi operativi attuali. Il ruolo di questa libreria è molto importante in settori come l'industria video ludica. Nei videogiochi è necessario sfruttare al massimo l'hardware grafico per creare giochi belli e coinvolgenti. I PC dei videogiocatori tuttavia sono limitati in potenza. Oggigiorno il miglioramento dell'hardware non è più veloce come un tempo. Non potendo più fare affidamento su di questo, l'industria deve adottare tecnologie software innovative che permettano un maggiore controllo per ottenere il massimo dalla macchina.

Il progetto si allinea a questo pensiero e sfrutta l'API per implementare all'interno del motore, un sistema di rendering multithread. Il multithreading permette di utilizzare al meglio le CPU moderne e massimizza l'utilizzo delle GPU. Questo è possibile solo con API di nuova generazione come Vulkan, prima erano impossibili con API come OpenGL. Lo scopo per cui il motore viene realizzato è infine permettere la creazione di demo sperimentali in modo sempllice e veloce. Tramite questo motore, un programmatore è in grado di specificare agilmente modelli e texture, assemblare con essi oggetti 3D e renderizzarli in una scena. L'obiettivo finale della tesi è creare una demo utilizzando questo motore e misurare le sue prestazioni.

La tesi è organizzata in capitoli il cui contenuto è qui brevemente riassunto.

- Capitolo 1: nel primo capitolo si introduce il lettore nel contesto sociale ed economico in cui si cala questo progetto e la tecnologia su cui si basa.

- Capitolo 2: il secondo capitolo parla dell'API Vulkan, introducendola attraverso il suo predecessore: OpenGL. Le due librerie

vengono messe a confronto evidenziando i punti di forza e i punti di debolezza.

- Capitolo 3: il terzo capitolo offre un'analisi tecnica della libreria Vulkan. Vengono approfonditi nei dettagli alcuni funzionamenti chiave dell'API, nonché la sua struttura. Questo capitolo è indispensabile in quanto propedeutico alla comprensione della progettazione del motore nel capitolo 4.

- Capitolo 4: il quarto capitolo espone il processo che ha portato alla realizzazione del motore. Si espongono gli obiettivi e le problematiche che sono state affrontate. Si illustra l'architettura e tutti i dettagli del design degni di nota. Infine vengono inclusi e commentati alcuni ritagli rilevanti del codice implementativo.

- Capitolo 5: il quinto capitolo introduce allo sviluppo della demo e ne misura i risultati. Viene mostrato un esempio di come realizzare una scena 3D di test poi si misurano i risultati abilitando o disabilitando il rendering multithread. Infine si fa una comparazione con una demo OpenGL.

- Capitolo 6: il sesto capitolo contiene una breve riflessione sul lavoro svolto e le prospettive di questo progetto per futuri sviluppi.

Indice

1 Computer Grafica e campi di applicazione **7**
1.1 Nuove API per i videogiochi PC 8
1.2 Mobile gaming 10

2 Perché Vulkan ? **13**
2.1 Cos'è Vulkan 13
2.2 Da OpenGL a Vulkan 13
2.2.1 OpenGL 13
2.2.2 Vulkan 17
2.2.3 Confronto 19
2.3 Indicazioni e Controindicazioni 20

3 Analisi dell'API Vulkan **21**
3.1 Struttura generale 21
3.2 I principali oggetti Vulkan 23
3.2.1 L'istanza Vulkan 23
3.2.2 Dispositivo logico e dispositivo fisico 24
3.2.3 La Swap Chain 25
3.2.4 Immagini e viste sulle immagini 26
3.2.5 Renderpass 26
3.2.6 La Pipeline 27
3.2.7 Descriptor Sets per gli Shader 28
3.2.8 Push Constant 29
3.2.9 Buffer di comando 30
3.2.10 Oggetti per la Sincronizzazione del codice . . . 31
3.2.11 Trasferimenti dati e Barriere di memoria 33
3.2.12 Panoramica 36
3.3 Estensioni, utility e tool 37
3.3.1 Validation Layers 37
3.3.2 RenderDoc 37
3.3.3 SPIR-V Cross 37

3.3.4 Vulkan Samples 38
3.3.5 Vulkan Memory Allocator 38
3.4 Supporto 39
3.4.1 GPU info 39

4 Sviluppo di un motore di rendering in Vulkan 41
4.1 Analisi 41
4.1.1 Obiettivi 41
4.1.2 Problematiche 44
4.2 Progettazione 45
4.2.1 Architettura 45
4.2.2 Dettagli del Design 49
4.3 Implementazione 58
4.3.1 Ambiente di lavoro 58
4.3.2 Metodologia 59
4.4 Dettagli implementativi 61
4.4.1 Utilizzo dei Validation Layers 61
4.4.2 Registrazione dei buffer di comando su più thread 63
4.4.3 Shaders 66

5 BenchMarks 71
5.1 Obiettivo della Demo 71
5.1.1 Parametri di misura 71
5.1.2 Metodo di lavoro 72
5.2 Creazione di uno scenario sintetico 72
5.2.1 Progettazione 72
5.2.2 Implementazione della demo 74
5.3 Analisi delle prestazioni dell'Engine 76
5.4 Confronto con demo OpenGL 78

6 Conclusioni e sviluppi futuri 81

Capitolo 1

Computer Grafica e campi di applicazione

Al giorno d'oggi il mondo è pervaso da innumerevoli contenuti digitali. Ogni forma d'arte, ogni attività umana, utile o dilettevole che sia, necessita ormai di possedere una sua rappresentazione digitale. La digitalizzazione permette di veicolare l'informazione, superando il mezzo fisico. Tramite la rete internet i contenuti circolano alla velocità della luce per arrivare ovunque nel mondo. Le piattaforme online per la condivisione, in primis i social network, condensano il flusso informativo. Sia da un punto di vista commerciale economico che culturale e sociale, il mondo sembra non poter fare a meno di internet e dei contenuti digitali. Inoltre, grazie a piattaforme fisiche come gli smartphone, questi contenuti sono sempre a portata di mano, e non ci abbandonano mai. Che si tratti di una azienda che deve vendere un prodotto, un'artista che vuole diffondere le sue creazioni, o uno spettatore che vuole solo intrattenersi, il medium digitale è irrinunciabile.

Fruire dei contenuti digitali ci porta a fare esperienza di una nuova dimensione, che sfugge ai parametri del mondo reale. Grazie all'evoluzione congiunta di hardware e software; negli anni si è riusciti a creare prodotti digitali sempre più belli, complessi e raffinati. Il livello di complessità delle macchine che teniamo in mano è ignorato dalla maggior parte di noi. La tecnica è talmente complessa che ai più sembra quasi magia. Questa magia è ancora più incredibile quando diventa interattiva. Medium quali i videogiochi e le simulazioni 3D ci permettono di interagire con un mondo fittizio che asseconda i nostri desideri. Esperienze impossibili o costosissime da vivere nella realtà, possono oggi essere ricreate al computer con una fedeltà ed un coinvolgimento sempre maggiore. Oggi si parla di realtà aumentata, realtà mista e realtà virtuale. Visori, assieme a speciali metodi per l'input, ci

permettono di vivere esperienze nuove. Possiamo concederci di vivere esperienze in mondi il cui unico limite è la nostra fantasia. Possiamo persino fondere la realtà con i prodotti digitali e creare infinite possibilità.

Tutto questo è permesso grazie a complessi calcolatori. Ma la vera invenzione, che ha creato il concetto di mondo virtuale è la grafica a computer. Un insieme di geniali stratagemmi matematici per poter generare immagini a schermo che riproducono il modo che abbiamo noi di osservare il mondo. É assurdo pensare come basti moltiplicare qualche matrice per trasformare sterili numeri in bellissime scene tridimensionali.

Ma quanto belle e complesse possono diventare queste scene virtuali? Il vero limite per fruire di una scena 3D in tempo reale è individuabile nel dispositivo che la rende. La matematica ci descrive come calcolare la scena, ma ciò che svolge il lavoro è il calcolatore. Produrre calcolatori sempre più potenti è un obbiettivo costante di ogni azienda del settore hardware. Tuttavia, per vincere la gara delle prestazioni e fare grafica di migliore qualità, non basta un'auto che sia semplicemente più potente. Per vincere serve anche un pilota esperto, che individui il percorso migliore sul tracciato, che scelga le gomme migliori, e gestisca in modo ottimizzato le fermate ai box. Stiamo parlando di grafica ad alte prestazioni.

Per ottenere il massimo dalla macchina, è necessaria una tecnologia che permetta allo sviluppatore di avere massimo controllo, per massime prestazioni. Vulkan è stato introdotto nel 2016 per dare una risposta alla domanda di nuovi regimi prestazionali in ambito real-time. Negli ultimi 3 anni, dal momento che è stata scritta questa tesi, l'API si è diffusa su tutte le maggiori piattaforme ed è il nuovo standard platform-indipendent che si sta imponendo sul mercato.

1.1 Nuove API per i videogiochi PC

API che possano sfruttare a pieno le risorse hardware sono vitali in scenari in cui non è possibile scalare con hardware più potente. Questo accade molto spesso nel rendering real-time di scene interattive. I videogiochi per esempio devono girare su hardware economico, compatto e disponibile al comune giocatore. Per tale motivo, un utilizzo ottimale delle risorse è fondamentale per la buona riuscita di un prodotto video-

ludico. Ogni video gioco presenta sempre indicazioni sulla piattaforma hardware e software compatibile ed una serie di requisiti minimi. Tali requisiti minimi comprendono: la tipologia del processore, la memoria installata, spazio su disco, scheda video minima supportata. Questi requisiti vengono stimati dal team di sviluppo che a volte deve prendere scelte difficili. Molti video giocatori attendono con trepidazione l'uscita del prossimo videogioco della saga X e vorrebbero che la grafica fosse ancora più bella ed immersiva del capitolo precedente. Gli sviluppatori del gioco lo sanno e vorrebbero aumentare la resa visiva del proprio gioco; così da poterlo pubblicizzare in modo accattivante. Tuttavia il gioco deve anche essere accessibile ad un bacino di utenza che sia esteso per garantire un buon guadagno dalle vendite. E' necessario che i requisiti minimi rimangano bassi così da permettere a tutti di giocare. Per chi invece possiede schede potenti, il gioco non solo deve funzionare ma deve farlo bene. All'occhio piace la qualità visiva, ma per essere giocato, il gioco, deve stare su un numero sufficiente di fotogrammi al secondo. Su PC, un gioco dovrebbe riuscire a girare ad un frame rate stabile non inferiore ai 60 fotogrammi per secondo. A questo ritmo la scheda video produce frame alla stessa velocità di refresh dei comuni monitor per computer. Avere il frame-rate pari al refresh-rate del monitor evita problemi come lo screen-tearing: che spezza l'immagine in fasce orizzontali se ci sono troppi fotogrammi; oppure lo stuttering, quando gioco sul monitor va a scatti a causa di pochi fotogrammi.

Un caso particolare sono i giochi e le simulazioni in realtà virtuale. Applicazioni che eseguono sui visori devono produrre il doppio dei fotogrammi rispetto che su un normale monitor. Questo accade poiché la scena deve essere resa su due monitor con due prospettive differenti. In più, il numero di fotogrammi per secondo deve stare ben oltre i 60, solitamente oltre i 90. Quando si indossa il visore ci si tende a muovere normalmente. Nella realtà i nostri occhi sono abituati a vedere lo spazio 3D in modo fluido, tale fluidità deve essere garantita anche dentro al visore. Un gioco in realtà virtuale con un frame rate basso, non è solo fastidioso da giocare, può indurre anche una forte nausea.

Ormai dovrebbe essere chiaro che chi lavora in questa industria deve continuamente combattere per poter produrre giochi di qualità. Nel settore videoludico PC Vulkan e DirectX12 sono le 2 API di riferimento. L'utilizzo di DirectX12 è limitato alle macchine Windows10

mentre Vulkan opera anche su Linux, Android e persino su Macintosh grazie alla libreria MoltenVk. Le console come la PlayStation 4 di Sony invece, non le supportano, in quanto lavorano con API proprietarie.

Un ultimo aspetto da notare è che per lo scenario PC, applicazioni 3D tendono ad utilizzare Vulkan per guadagnare più in qualità che in risparmio energetico. Solitamente i sistemi PC sono alimentati a corrente oppure sono dotati di potenti batterie. Gli utenti sono più interessati alla qualità dell'esperienza che al risparmio energetico. Questa logica si capovolge totalmente se parliamo di sistemi mobile.

1.2 Mobile gaming

Android ed Apple hanno pienamente rivoluzionato il concetto di cellulare che si aveva prima del 2008. Ora tra le mani abbiamo dei computer a tutti gli effetti, le chiamate sono forse l'aspetto meno importante quando si compra un telefono al giorno d'oggi. Gli smartphone, che non sono altro che computer tascabili, possono oggi accedere grazie ai negozi digitali a miriadi di applicazioni. Sui cellulari vengono installate le ultime innovazioni nel campo dei sensori, processori, memorie e materiali di costruzione. Sono dispositivi che cavalcano l'innovazione tecnologica, a volte veri e propri status symbol, per i quali la clientela è disposta a pagare prezzi a quattro cifre. Ciò che attira così tanti utenti a dare importanza agli smartphone è la loro versatilità e semplicità di utilizzo, unita ad un parco App immenso. Le piattaforme smatphone attraggono quindi sia clienti che sviluppatori, creando un mercato che oggi è molto florido.

La potenza hardware di questi dispositivi spinge moltissimi sviluppatori a buttarsi nello sviluppo di applicazioni computazionalmente costose come i video giochi 3D. Questi giochi però differiscono notevolmente da quelli presenti su PC e Console. Il videogioco PC è pensato per essere giocato da seduti, davanti ad un monitor di buone dimensioni, con dispositivi adatti all'input: come una tastiera, un mouse o un joypad. Una sessione di gioco richiede tempo, dei salvataggi manuali, ed un ambiente tranquillo che massimizzi il godimento del prodotto artistico. I giochi mobile invece sono immediati, non richiedono salvataggi, sono fatti per utilizzare lo schermo sia come input che come output e possono essere giocati ovunque ci si trovi. I giochi sono spesso fatti per una fruizione di breve durata, sono per lo più dei passatempi. Il target è perciò un utente che apre il gioco per ingannare

il tempo, magari quando è fuori casa e non sa che fare. Motivo per cui giochi tendono ad essere semplici, sia nel gameplay che nell'aspetto tecnico.

Per quanto riguarda il bacino di utenza, questo è molto più ampio della contro parte PC. Non tutti infatti possiedono un PC da gaming o una console, mentre tutti hanno uno smartphone. Inoltre i giochi per smartphone costano generalmente molto meno dei giochi PC e Console, ne esistono anche di molti gratuiti che guadagnano con banner pubblicitari.

Chi sviluppa questi titoli solitamente non può puntare troppo sull'impatto visivo del video gioco. Lo schermo piccolo e l'input tramite touchscreen non permettono una buona immersione. Inoltre l'hardware dell'utenza è estremamente vario. Si va dai più potenti ammiragli delle case produttrici, fino alla fascia bassa entry-level degli smartphone più economici. Si punta perciò a giochini veloci e semplici che tengano incollato il giocatore, portandolo a riaprire il gioco il più spesso possibile.

Tuttavia queste applicazioni non possono fare a meno di girare in maniera ottimizzata. La piattaforma cellulare ha si il vantaggio della versatilità, ma porta con sé il contro di poter contare su una batteria molto limitata. Un cellulare deve arrivare sempre, come minimo, a fine giornata. Un'applicazione grafica deve assicurarsi di consumare meno batteria possibile. Nessuno avrebbe piacere di giocare ad un gioco al cellulare se pochi minuti gli costassero metà della sua batteria. Solitamente sui telefoni si è sempre usato OpenGL, ma OpenGL sfrutta un solo core CPU per lavorare. Gli smartphone di oggi sono dispositivi votati al multithreading e multitasking. Montano processori con molti core. Vulkan quindi è la soluzione migliore per applicazioni grafiche su dispositivi Android. Con Vulkan è possibile sfruttare l'hardware mobile al meglio e diminuire notevolmente i consumi energetici. Infatti distribuendo il lavoro su tutti i core la CPU potrà girare ad un clock inferiore, consumando meno. Ridurre i consumi è vitale per prolungare la durata della batteria e migliorare l'esperienza utente nell'utilizzo di applicazioni grafiche.

Capitolo 2

Perché Vulkan ?

2.1 Cos'è Vulkan

Vulkan è una API cross-platform a basso overhead per il rendering 3D e la computazione. E' stata per la prima volta annunciata alla conferenza GDC del 2015 dal Khronos Group: un gruppo di lavoro no-profit, finanziato da molti tra i più grandi produttori di software e hardware sul mercato. Il gruppo da anni mantiene standard diffusi come OpenGL e WebGL. Nel febbraio 2016 viene rilasciato il kit di sviluppo open source e la versione 1.0 delle specifiche.

In questi ultimi anni Vulkan si è diffuso sulla gran parte delle piattaforme hardware e software. Molti motori di rendering realtime hanno adottato l'API per lo sviluppo di videogiochi e applicazioni 3D sia in ambiente desktop che mobile. Vulkan si accosta alle nuove API di sviluppo 3D, al pari di DirectX12 e Metal. Queste API permettono di sfruttare meglio le risorse hardware. Ciò si traduce in migliori prestazioni e minori consumi, grazie ad un maggiore controllo sull'hardware. Tuttavia Vulkan si differenzia in quanto mantiene la filosofia cross-platform del suo predecessore OpenGL; essa è slegata ed indipendente dal sistema operativo, disponibile in moltissimi linguaggi e supportata da pressoché tutto l'hardware recente.

2.2 Da OpenGL a Vulkan

2.2.1 OpenGL

Forse non è corretto dire che Vulkan sia il successore di OpenGL, ma sicuramente OpenGL è il suo predecessore. OpenGL, nel corso degli anni ha fatto la storia come caposaldo della programmazione 3D cross platform. La storia di OpenGL è molto lunga, basti pensare che la

prima versione 1.0 venne rilasciata nel 1992. Da allora il controllo sullo sviluppo dell'API è passato di mano svariate volte. Negli anni le innovazioni sono state inserite tramite estensione, portando il codice ad avere radici molto profonde. É inevitabile notare come in tutti questi anni la tecnologia abbia fatto passi da gigante in tutti i campi, e la computer grafica non è stata da meno. L'hardware, in costante evoluzione, ha introdotto sempre nuove possibilità. Un esempio è il passaggio dall'utilizzo di pipeline statiche a pipeline programmabili. Anni fa le schede video erano più semplici rispetto ad oggi, con capacità molto limitate, in quanto molta della logica di rendering era scritta nell'hardware. I driver quindi, offrivano al sistema interfacce molto più semplici di ora per il controllo della scheda. Il fatto che OpenGL stessa sia nata come API di alto livello, ben si adatta al vecchio modo di interfacciarsi con l'hardware grafico.

Col tempo però le schede si sono evolute, aumentando la complessità di utilizzo e il numero di funzionalità disponibili. OpenGL si è evoluta di conseguenza, introducendo nuove funzionalità e al tempo stesso deprecando vecchie funzioni. Tuttavia ha sempre mantenuto un approccio di alto livello sulla macchina. Come vedremo a breve, questo comporta diversi problemi.

I produttori hardware, anno dopo anno, GPU dopo GPU, hanno sempre dovuto rendere disponibili le funzionalità delle loro schede grafiche ad una API di alto livello. La conseguenza diretta di questo sono dei driver molto complessi. I driver per OpenGL devono fare ben più di un semplice interfacciamento col silicio. É infatti necessario che questi driver implementino le loro ottimizzazioni per la scheda fisica che gestiscono. Per far si che OpenGL viaggi il più velocemente possibile su qualsiasi scheda, integrata o dedicata che sia, i produttori devono scrivere codice che sia in grado di ottimizzare il lavoro astratto dell'API.

Uno dei problemi principali che si nota è lo scarso apporto informativo dell'applicazione OpenGL in fase di inizializzazione. OpenGL durante il caricamento iniziale non fa quasi nulla per informare il driver. Per esempio: quali risorse utilizzerà l'applicazione, in che modo e in quale stage della pipeline ...ecc. OpenGL non può e in un certo senso non vuole farlo; essere una API di alto livello richiede questo: non preoccuparsi dell'hardware. Tuttavia ottenere prestazioni decenti non è in discussione. Il driver perciò, a seconda dell'implementazione, dovrà tramite euristica trovare l'ottimizzazione corretta per l'app.

Fondamentalmente stiamo parlando di tirare a indovinare nei casi peggiori. Pur non essendo il modo migliore per risolvere il problema, i driver OpenGL svolgono un egregio lavoro e permettono agli sviluppatori di fare grafica con relativa agilità. Potersi concentrare su "cosa fare" invece che su "come poterlo fare" è un enorme vantaggio.

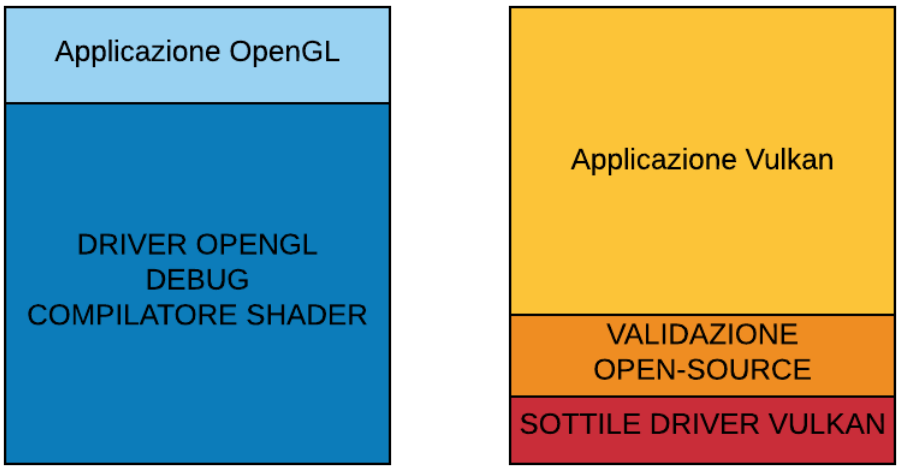

Figura 2.1: Due API, due modi di distribuire la complessità.

Allora perchè abbiamo bisogno di API come Vulkan? Ebbene se è vero che i driver risolvono il problema lato GPU, sussiste sempre un'altro limite. Finora non se n'è parlato ma è ora di introdurre la CPU. L'unità di elaborazione centrale è il luogo in cui il codice applicazione esegue; nel particolare del nostro discorso, è dove l'app e il driver assemblano comandi e istruzioni da inviare alla scheda video. Ora pensiamo ad un comune task di rendering. La CPU controlla lo stato dell'applicazione, calcola le trasformazioni per il prossimo fotogramma, imposta i comandi di disegno e poi chiama i driver, il driver fa il suo lavoro e invia dati e comandi al bios della scheda. Ora immaginiamo che il rendering dell'immagine sia semplicissimo e la scheda termini il lavoro molto rapidamente. Mettiamo caso che la GPU svuoti la coda di comando più velocemente di quanto l'applicazione riesca a riempirla. La GPU si ritroverebbe in stallo, costretta a non far nulla in attesa che l'applicazione si decida a sottomettere il prossimo frame. Questo problema, che può sembrare assurdo, è un invece presente in molte applicazioni realtime. Scene complesse possono richiedere più tempo per essere definite che per essere rese graficamente. É come quando perdiamo giorni a cercare su internet il negozio che ci fa il

prezzo più basso e poi una volta che ci siamo decisi compriamo con un click in pochi secondi.

Per ovviare a questo problema dobbiamo introdurre il concetto di calcolo parallelo. Una buona pratica della programmazione è dividere il lavoro su più thread, soprattutto quando ci si trova davanti a scenari detti "embarrassingly parallel". Ovvero il task è molto facile da dividere in molteplici flussi di lavoro che richiedono minima sincronizzazione. Per avere più thread servono più core e perciò nell'ultimo decennio l'industria manifatturiera di CPU si è orientata verso l'incremento costante del numero dei core. OpenGL però è nata molti anni prima che i processori multicore si diffondessero. Il cuore della libreria è organizzato intorno ad un contesto di variabili, aggiornate da seguenti chiamate all'API. Invocare OpenGL da più flussi sullo stesso contesto del processo applicativo provoca malfunzionamenti o crash. L'unico modo per OpenGL di scalare in prestazioni CPU-side è girare su un processore con core più potenti e veloci.

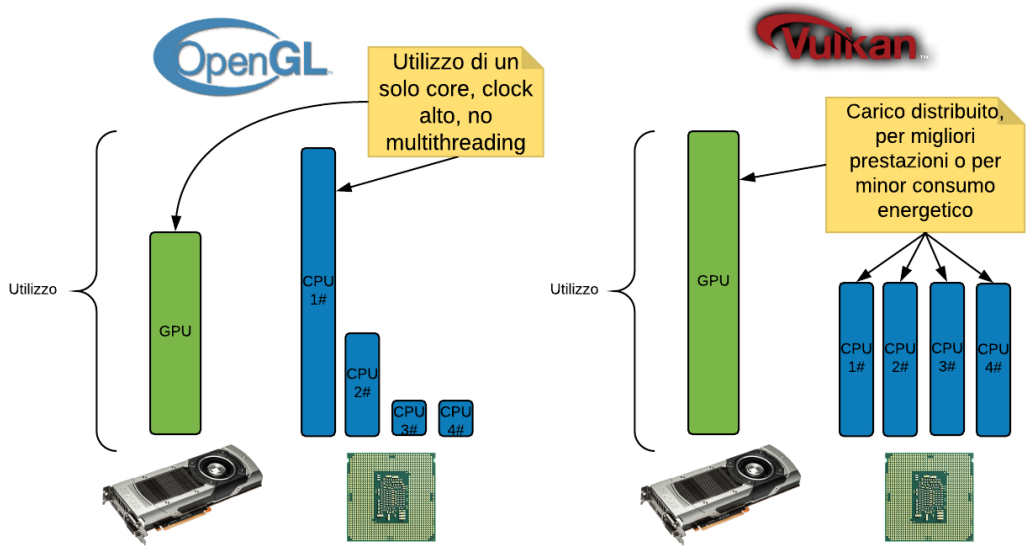

Figura 2.2: Vulkan permette di massimizzare l'utilizzo della GPU grazie al lavoro congiunto di tutti i thread CPU. Inoltre aiuta a minimizzare il clock cpu massimo, contenendo il consumo energetico.

Per anni la legge di Moore ha spinto le aziende ad incrementare costantemente il numero di transistor nei core tramite miniaturizzazione. In tal modo si riducono consumi, emissioni di calore e si può aumentare il clock. Ormai da tempo però l'industria si è accorta che questo trend è destinato a rallentare e bloccarsi. Siamo ormai arrivati al limite della miniaturizzazione, non si può certamente scendere sotto all'atomo, inoltre non è possibile incrementare eccessivamente il clock. A velocità troppo alte basterebbe una curva microscopica su una pista bus, per far rallentare i bit sull'esterno della curva e causare errori di lettura. Da anni quindi si cerca di aumentare le prestazioni aumentando il numero dei core. OpenGL è destinata perciò a rimanere indietro poiché non può cavalcare il nuovo trend di sviluppo. Al giorno d'oggi sfruttare il multithreading ove possibile è la pratica comune per applicazioni intensive desktop, server e mobile; in pratica ovunque.

OpenGL mantiene comunque i suoi vantaggi e rimane un ottimo strumento per sviluppare applicazioni semplici, dove non sono richieste prestazioni massime. Ma il mondo non è fatto solo di cose semplici e per superare la prova del futuro serve un software fresco, all'avanguardia, che sappia tirare fuori il meglio dalla macchina.

2.2.2 Vulkan

Vulkan nasce dalle ceneri di Mantle 1.0, un progetto AMD che non è mai realmente decollato. Mantle era una API ideata da AMD e DICE nel 2013 che prometteva tecniche di rendering e prestazioni rivoluzionarie. Il problema stava nel supporto. Siccome Mantle era un progetto AMD, non avrebbe mai ricevuto il supporto sulle schede del competitor rivale, Nvidia. Viste le scarse prospettive AMD cede Mantle al Khronos Group, il quale nel 2015 annuncia Vulkan. Il Khronos Group prende il meglio di Mantle e lo sviluppa nell'API del futuro. Vulkan diventa immediatamente il nuovo standard per scrivere codice grafico ad alte prestazioni, capace di funzionare su ogni piattaforma.

Quando ci si accinge ad introdurre Vulkan, bisogna inevitabilmente scontrarsi contro la sua difficoltà. L'API infatti è tutt'altro che semplice da usare, uno stacco completo da OpenGL in questo senso. Il nuovo paradigma è: massimo sforzo per massima resa. Dal codice Vulkan traspare la complessità della macchina, il programmatore è costretto a fare i conti con l'hardware. Vulkan non fa sconti, il suo

target sono applicazioni 3D ad alte prestazioni, ma è solo uno strumento; qualsiasi ottimizzazione la fa lo sviluppatore dell'applicazione a sue spese. Questa trasparenza viene anche accompagnata da una scelta di design importante. Vulkan, contrariamente ad OpenGL non include uno stato di default. Lo sviluppatore è chiamato, a tempo di inizializzazione, a prendere una decisione esplicita e precisa su ogni funzionalità base che vuole o non vuole usare. Non solo; Vulkan delega allo sviluppatore il compito di cercarsi l'hardware compatibile, verificare che il driver presenti le estensioni utili alla sua applicazione e procedere di conseguenza. Questo è solo un esempio ma se ne potrebbero fare tantissimi altri; ciò che è importante notare è che Vulkan vuole raccogliere maggiori informazioni possibili subito, per risparmiare lavoro al driver durante il rendering. Un altro effetto positivo dell'essere espliciti è che il programmatore ha più controllo sull'applicazione. Con tanti parametri da impostare, dimenticarsi qualcosa può succedere; ciò porta nella maggior parte dei casi ad un crash, o se si è sfortunati, ad un comportamento indeterminato dell'applicazione.

Per affrontare problematiche di debug, Vulkan impiega un sistema avanzato e molto innovativo. Un'applicazione Vulkan durante la normale esecuzione non compie alcun controllo di errore, è liberissima di andare in crash se qualcosa va storto. Se però uno sviluppatore è interessato a fare debug può attivare a tempo di inizializzazione i livelli di validazione. Questi livelli possono essere caricati sotto forma di estensioni e si interpongono tra le chiamate a funzione e la mappatura al driver. Il programmatore può attivarne quanti ne vuole, solitamente sono tematici, ognuno controlla una parte di funzionalità. Chiaramente attivare questi livelli impatta enormemente le prestazioni ma offre un debug notevolmente superiore ad OpenGL. In più, quando non si sta facendo debug l'app potrà fare a meno di perdere tempo sul controllo di errore, risultando più efficiente.

Nella sottosezione 2.2.1, si è parlato delle limitazioni causate dal contesto singolo di OpenGL. Vulkan supera questo concetto di design. Non avendo un contesto, non presenta uno stato ed è perciò interrogabile da più flussi di istruzioni contemporaneamente senza creare problemi. Come tante altre caratteristiche dell'API che offrono un eccellente grado di libertà questa porta con se i suoi costi. Come sempre, da grandi poteri derivano grandi responsabilità. OpenGL manteneva uno stato per assicurare coerenza al susseguirsi di chiamate alla libreria. Con Vulkan il programmatore ha il controllo e deve egli stesso

gestire lo stato della sua applicazione. Per permettere ciò Vulkan offre al programmatore tre strumenti di sincronizzazione, approfonditi nel capitolo successivo. In Vulkan gestire la sincronizzazione è vitale. É importante gestire la sincronizzazione GPU-CPU , GPU-GPU e CPU-CPU. Questi sono solo alcuni degli elementi che rendono Vulkan un'API difficile, complessa e verbosa. Ma, se lo sviluppatore accetta di pagare questo prezzo, avrà la possibilità di implementare le più moderne tecniche di rendering e spremere le prestazioni dell'hardware come mai prima d'ora.

2.2.3 Confronto

Per dare una rappresentazione riassuntiva di quello che è stato detto finora; propongo di seguito la tabella 2.1. In tale rappresentazione vengono affiancate le opposte scelte di design tra Vulkan ed il suo predecessore.

	
Progettata per vecchie workstation con rendering diretto.	Progettata per hardware d'avanguardia, comprese le piattaforme cellulari.
Singolo contesto, singola macchina a stati.	Nessun contesto globale, strutturata ad oggetti.
Lo stato è globale e legato al contesto.	Gli stati sono localizzati nei buffer di comando.
Solo operazioni in sequenza.	Operazioni parallele sono possibili.
Gestione della memoria e sincronizzazione nascosta.	Esplicito controllo sulle allocazioni e sulla sincronizzazione.
Controllo degli errori esteso e sempre attivo.	Nessun Controllo, ma c'è la possibilità di agganciare livelli di validazione.
Compilatore degli shader incorporato nel driver.	Compilatore SPIR-V esterno, garantisce maggiore flessibilità ed affidabilità.
Driver complesso, consistenza cross-platform difficile da garantire.	Driver semplice, alta predicibilità del codice applicazione

Tabella 2.1: Un riepilogo sul differente design.

2.3 Indicazioni e Controindicazioni

Per quanto Vulkan rappresenti lo stato dell'arte, non sarebbe saggio in questa tesi, né in altro luogo, affermare che Vulkan sia la migliore soluzione per lo sviluppo di applicazioni 3D. Abbiamo già parlato di come Vulkan sia una pessima scelta per progetti semplici. Ma questo non è solamente dovuto alla verbosità della libreria. La scelta se utilizzare Vulkan per il proprio progetto deve essere meditata, non si tratta solo di doversi sobbarcare più lavoro. Dando per scontato che scrivere più codice non sia un problema, di seguito elenco scenari in cui è bene usare Vulkan.

Scegliere Vulkan quando:

- Le prestazioni dell'applicazione sono limitate dalla CPU.
- L'applicazione potrebbe guadagnare in prestazioni grazie ad una gestione ad hoc della memoria e della sua sincronizzazione.
- L'applicazione richiede ottimizzazioni per un hardware specifico.
- Si è in possesso di conoscenze sufficienti ad implementare una corretta gestione delle allocazioni ed una buona sincronizzazione.

Evitare Vulkan quando:

- Le prestazioni sono limitate dalla potenza della GPU.
- Non c'è necessità di applicare ottimizzazioni avanzate sulla gestione della memoria.
- Non si è sicuri di essere in grado di applicare ottimizzazioni migliori di quelle già automatizzate da API di alto livello.

Sussiste sempre e in ogni caso come ottima motivazione lo scopo formativo. Vulkan permette di ampliare considerevolmente le proprie conoscenze in campo grafico tecnico. É al giorno d'oggi una importante risorsa, che non può rimanere estranea a chi vuole diventare un professionista in materia.

Capitolo 3

Analisi dell'API Vulkan

3.1 Struttura generale

Vulkan presenta una macro architettura a livelli ed alcuni di questi sono facoltativi. Il livello più alto è il luogo del codice applicazione. Da lì ogni chiamata a funzione viene diretta al runtime di Vulkan, il cosiddetto loader. Il Vulkan Loader è una importante interfaccia, distribuita assieme ai driver Vulkan[2]. Questa libreria incanala le chiamate a funzione verso i livelli sottostanti. Al di sotto infatti, possono essere presenti molti driver, grazie ai quali Vulkan può operare sull'hardware. Questi moduli chiamati ICD, acronimo per "Installed client driver", contengono ognuno una implementazione dell'API per uno specifico dispositivo fisico. Il loader, oltre che indirizzare le chiamate al corretto ICD, permette di abilitare al suo interno dei livelli opzionali per il controllo degli errori.

Il Vulkan Loader, mantenuto da Khronos Group, è open source ed'è disponibile su GitHub; i livelli di validazione sono solitamente distribuiti assieme ai kit di sviluppo. Questi livelli intermedi di validazione possono anche essere liberamente implementati dal programmatore dell'applicazione per un debug personalizzato. Quando uno o più livelli di validazione sono attivi, il loader rimbalza le chiamate alla libreria sui livelli di validazione che gli competono; poi ritornano al loader per essere mappate sugli ICD. Tutta questa intermediazione crea un overhead minimale e trascurabile. Però se si vuole, si può bypassare il Loader per parlare direttamente con l'ICD desiderato. É possibile infatti, richiedere a runtime l'indirizzo della function call dell'ICD desiderato e mapparla manualmente ad una funzione Vulkan. In tale modo si è sicuri di eliminare qualsiasi overhead. Le figure 3.1 e 3.2 mostrano le differenti configurazioni.

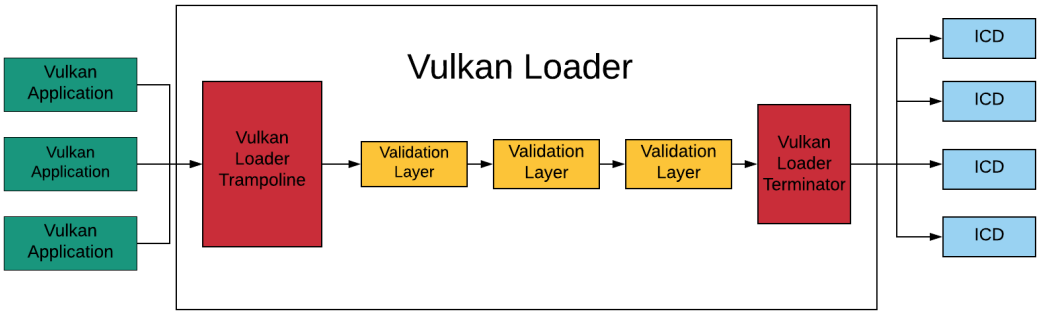

Figura 3.1: Percorso di una chiamata a funzione con mappatura automatica.

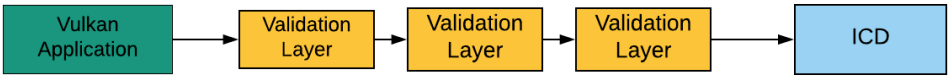

Figura 3.2: Percorso di una chiamata a funzione con mappatura manuale.

Quando si decide di gestire la mappatura delle chiamate e caricare ogni singola funzione lo si può fare sia a livello di istanza che di dispositivo. Vulkan offre 3 tipi di funzioni[2].

- **Funzioni di livello globale:** compiono azioni sulla libreria, ma senza indirizzare una specifica istanza.
- **Funzioni di livello istanza:** compiono azioni rivolte ad oggetti subordinati ad una specifica istanza ma non legati ad un dispositivo.
- **Funzioni di livello di dispositivo:** compiono azioni dirette ad oggetti legati ad un dispositivo.

Utilizzando la funzione *vkGetInstanceProcAddr* l'utente ottiene una mappatura diretta di una funzione di livello istanza con l'istanza utilizzata. Per mappare una funzione rivolta ad un dispositivo, bisogna invece utilizzare *vkGetDeviceProcAddr*. Una volta caricate le funzioni di libreria l'applicazione può cominciare ad utilizzare Vulkan.

Tutte le applicazioni Vulkan che mirano a renderizzare immagini e presentarle su una superficie, seguono la stessa sequenza di operazioni preliminari. In questa fase si creano e configurano tutti gli oggetti necessari al rendering. Nella prossima sezione vedremo in maggior dettaglio questi oggetti e la loro gerarchia in un'applicazione Vulkan.

3.2 I principali oggetti Vulkan

3.2.1 L'istanza Vulkan

Il primo oggetto che un'applicazione Vulkan crea è l'oggetto *VkInstance*. Questo oggetto rappresenta l'applicazione stessa. Sebbene un processo ne possa creare più di una, gli oggetti subordinati ad una istanza non possono interagire con altre istanze. Per questo motivo se ne crea una sola, dalla quale dipendono tutti gli oggetti creati successivamente. Solo per questo oggetto, qui sotto si mostra un esempio di come si crea un oggetto in Vulkan, vedi figura 3.3. Vulkan offre sempre una funzione per la creazione che richiede una struttura informativa. Prima si inizializzano tutti i campi della struttura, poi si chiama la funzione. In Vulkan i campi delle strutture corrispondo ad impostazioni, quindi vanno tutti esplicitamente inizializzati. Per l'istanza è sufficiente specificare le estensioni Vulkan che si vogliono abilitare e alcune informazioni accessorie di versione dell'applicazione.

```
1  // struttura con info sulla mia applicazione
2  VkApplicationInfo appInfo = {};
3  appInfo.sType = VK_STRUCTURE_TYPE_APPLICATION_INFO;
4  appInfo.pApplicationName = "Hello Triangle";
5  appInfo.applicationVersion = VK_MAKE_VERSION(1, 0, 0);
6  appInfo.pEngineName = "No Engine";
7  appInfo.engineVersion = VK_MAKE_VERSION(1, 0, 0);
8  appInfo.apiVersion = VK_API_VERSION_1_0;
9
10 // creo la struttura di info per la creazione dell'istanza di Vulkan
11 VkInstanceCreateInfo createInfo = {};
12 createInfo.sType = VK_STRUCTURE_TYPE_INSTANCE_CREATE_INFO;
13 createInfo.pApplicationInfo = &appInfo;
14
15 // Chiedo a glfw di cercare tutte le estensioni richieste per GLFW
16 std::vector<const char*> glfwExtensions = getRequiredExtensions();
17 createInfo.enabledExtensionCount = static_cast<uint32_t>(glfwExtensions.size());
18 createInfo.ppEnabledExtensionNames = glfwExtensions.data();
19 createInfo.enabledLayerCount = 0;
20
21 //Creazione dell'istanza di Vulkan
22 if (vkCreateInstance(&createInfo, nullptr, &instance) != VK_SUCCESS) {
23   throw std::runtime_error("failed to create instance!");
24 };
```

Figura 3.3: Creazione di una istanza Vulkan.[4]

In questo esempio richiedo alla libreria GLFW le estensioni da abilitare. Vulkan da solo non può comunicare col sistema operativo. É progettato in maniera da ignorare la piattaforma. Se vogliamo utilizzare Vulkan per presentare immagini in una finestra dobbiamo prima assicuraci che nel sistema siano presenti le estensioni Vulkan necessa-

rie. Sia nel progetto che nell'esempio in figura 3.3 utilizzo GLFW, la quale gestisce le chiamate al sistema per la creazione della finestra e ricerca le estensioni richieste per la presentazione.

3.2.2 Dispositivo logico e dispositivo fisico

Quando in Vulkan dobbiamo inviare comandi all'hardware, ciò avviene sempre attraverso un livello di astrazione chiamato dispositivo logico. Il dispositivo logico serve a Vulkan per offrire una interfaccia comune a tutti i dispositivi fisici. Questo è senz'altro l'oggetto più importante, con esso potremo per esempio: creare immagini, buffer, definire gli stadi della pipeline e caricare gli shader. Vulkan ci permette con una istanza di gestire quanti dispositivi vogliamo.

Prima di creare un dispositivo logico, abbiamo bisogno di scegliere un dispositivo fisico. É possibile gestire anche configurazioni multi-GPU tramite apposite estensioni come *KHR_device_group_creation*, che accorpa più dispositivi in uno e permette di utilizzare ad esempio le configurazioni SLI. La scelta del dispositivo fisico è fondamentale. L'applicazione dovrà interrogare la libreria sull'hardware disponibile e scegliere il dispositivo che soddisfi le proprie esigenze.
Per una applicazione che deve presentare immagini in una finestra, è necessario verificare che il dispositivo:

- sia una scheda grafica integrata o dedicata.
- offra una coda per comandi grafici di disegno.
- offra una coda per la presentazione di immagini e che sia compatibile col tipo di superficie da noi scelta.
- supporti le feature supplementari che si vogliono utilizzare.

Una volta individuati i dispositivi compatibili, si possono richiedere ulteriori dettagli sulle caratteristiche hardware, in modo da individuare il dispositivo più prestante. Per esempio: è consigliabile preferire sempre una scheda video dedicata ad una integrata. Scelto il candidato migliore, si procede a creare un *VkDevice* in un modo molto simile a come si è visto per l'istanza. In questo caso, bisogna passare alla struttura di creazione la lista di funzionalità del dispositivo fisico che si vogliono utilizzare e le estensioni Vulkan necessarie.

Ora si possono ottenere gli oggetti *VkQueue* che permettono di gestire le code di comando. Le schede video hanno solitamente diversi

tipi di code; per ogni tipo ce ne possono essere più di una. Per la maggior parte delle schede, la coda di tipo grafico supporta sia comandi di disegno, di presentazione e di trasferimento dati. Alcune schede però potrebbero avere code dedicate per la presentazione o per il trasferimento. Alcune schede Nvidia ad esempio offrono una coda esclusiva per il trasferimento dati. I comandi inviati a questa coda vengono eseguiti da una componente chiamata DMA, "Direct Memory Access". Questo controller esegue parallelamente al processore principale della GPU. Con esso è possibile effettuare trasferimenti tramite bus PCI-Express senza rallentare la coda di rendering. Questo è estremamente utile per effettuare caricamenti di dati, ad esempio di textures, mentre la GPU disegna. Per fare questo Vulkan permette di utilizzare le code diverse su thread diversi. Quando una coda termina il proprio lavoro, può generare un evento per segnalare l'applicazione.

3.2.3 La Swap Chain

Un elemento fondamentale per gestire la presentazione di immagini è la swapchain. Si tratta di una catena di immagini sulle quali la GPU stamperà i pixel. Ogni volta che viene completato un passaggio di rendering, una di queste immagini viene colorata. Ogni volta che viene inviato un comando di presentazione, l'immagine correntemente visualizzata sulla superficie viene sostituita con la successiva. Per poter aggiornare lo schermo in modo pulito è perciò necessario avere almeno 2 immagini nella catena. In questo modo la GPU scrive sull'immagine nascosta, quando questa è pronta, avviene la sostituzione. Quando si crea l'oggetto *VkSwapchainKHR* vanno specificate: il numero delle immagini che comporranno la catena, il formato dei pixel delle immagini, la loro risoluzione e la modalità di presentazione. Quest'ultima è l'impostazione più importante. Di seguito viene riportato un elenco di metodi disponibili:

- VK_PRESENT_MODE_IMMEDIATE_KHR: le immagini swapchain vengono immediatamente presentate appena vengono aggiornate. Questo può causare effetti indesiderati come il tearing.
- VK_PRESENT_MODE_FIFO_KHR: si usa una coda FIFO, l'immagine presentata è la prima, mentre la GPU inserisce l'ultima immagine renderizzata per ultima. Se la coda si svuota a causa di un ritardo, la swapchain aspetta la GPU. Allo stesso modo la

GPU si ferma se tutte le immagini sono in coda in attesa di essere presentate.

- VK_PRESENT_MODE_FIFO_RELAXED_KHR: come la normale FIFO con la differenza che se la coda si svuota la swapchain non aspetta la GPU. Anche qui si può notare del tearing.
- VK_PRESENT_MODE_MAILBOX_KHR: una coda FIFO che però viene comunque aggiornata anche se piena. Se tutte le immagini sono in coda in attesa, la GPU sovrascrive comunque la più vecchia.

La soluzione migliore, che evita stalli dell'hardware, è la modalità mailbox. Grazie ad essa si può implementare il triple buffering che elimina il tearing senza introdurre latenze.

3.2.4 Immagini e viste sulle immagini

Dopo aver creato la swapchain per un determinato dispositivo e superficie, Vulkan crea automaticamente le immagini (*VkImage*) della swapchain. A questo punto è necessario creare per ogni immagine una vista (*VkImageView*). In Vulkan non si manipola mai una immagine direttamente, lo si fa invece attraverso una vista. La vista permette di accedere in lettura e scrittura ad una immagine. Viene impiegata sia per i target di rendering come le immagini della swapchain, che per le textures. Per le textures è possibile indicare i livelli per le mipmaps, mentre per qualsiasi immagine è possibile specificare quale porzione verrà utilizzata e con quale formato pixel. Grazie al concetto di vista si è liberi di avere più render target sulla stessa immagine, come poter avere diverse textures campionate dalla stessa immagine.

3.2.5 Renderpass

Vulkan introduce il concetto di renderpass. Per Renderpass si intende il procedimento che porta alla scrittura di una immagine in memoria. Effettuare il renderpass è perciò basilare per poter effettuare rendering. Il renderpass, rappresentato da (*VkRenderPass*), è un oggetto che descrive uno schema di rendering che fa solo riferimento al tipo e numero di risorse utilizzate. Possiamo infatti specificare quanti output vogliamo avere. Se questi output corrisponderanno alle immagini della swapchain saranno output di colore. Esistono anche altri output

come lo Z buffer per effettuare il test di profondità. Per generare questi output sono necessari dei sotto passaggi, i subpass. Il renderpass si suddivide internamente in uno o più sotto passaggi. Ogni subpass rappresenta una iterazione di rendering che prende in input l'output del sotto passaggio precedente e produce un output a sua volta. Ogni sotto passaggio ha una sua pipeline di rendering e può agire su output diversi tramite riferimenti agli output definiti nel renderpass. Per i subpass vanno definite le dipendenze con gli stadi delle pipeline precedenti. Un subpass dovrà aspettare che chi lo precede sia giunto allo stadio specificato e a quale stadio l'output verrà rilasciato con quale tipo di accesso.

I subpass in Vulkan sono forse l'argomento più complesso e di difficile comprensione. In questo progetto si fa uso di un unico subpass che incorpora il test di profondità con lo Z buffer. L'utilizzo di più subpass è frequente quando si vogliono aggiungere effetti di post processing. Immaginiamo di voler aggiungere un effetto vignettatura ad una scena 3D. Invece di effettuare il rendering due volte e quindi scrivere e leggere 2 volte dalla memoria, utilizziamo 2 sottopassaggi. Prima disegniamo tutti gli oggetti della scena, poi richiediamo il cambio di subpass e inviamo un comando di disegno con uno shader speciale. Questo shader non avrà in input una geometria da disegnare ma l'output del subpass precedente. In tale modo potrà editare i pixel dell'immagine tramite coordinate UV ed ottenere l'effetto.

3.2.6 La Pipeline

La creazione della pipeline è uno dei procedimenti più verbosi e persino più pesanti da fare in Vulkan. La pipeline è divisa in stage, alcuni di questi sono interamente programmabili tramite shader (vedi figura 3.4); altri invece sono fissi, di questi si possono solo settare dei parametri. Una pipeline fa sempre riferimento ad un renderpass ed agisce in un preciso subpass al suo interno. Gli stage fissi vanno esplicitamente configurati prima della creazione. Per esempio va specificato che tipi di dati verranno forniti per vertice e quali algoritmi di blending usare... ecc. Per gli stage programmabili, Vulkan accetta compilati in bytecode SPIR-V. Gli shader perciò vanno precompilati prima del lancio dell'applicazione. Lo SPIR-V può essere prodotto sia utilizzando GLSL che HLSL come linguaggi per lo shading. La pipeline Vulkan permette di impostare delle costanti dette "di specializzazione". Hanno una

funzione simile alle *#define* in C e C++, permettono di inizializzare dei parametri e modificare il comportamento dello shader appena prima della creazione della pipeline. Inoltre è possibile specificare anche la funzione entry point così da permettere un branching all'interno dello shader.

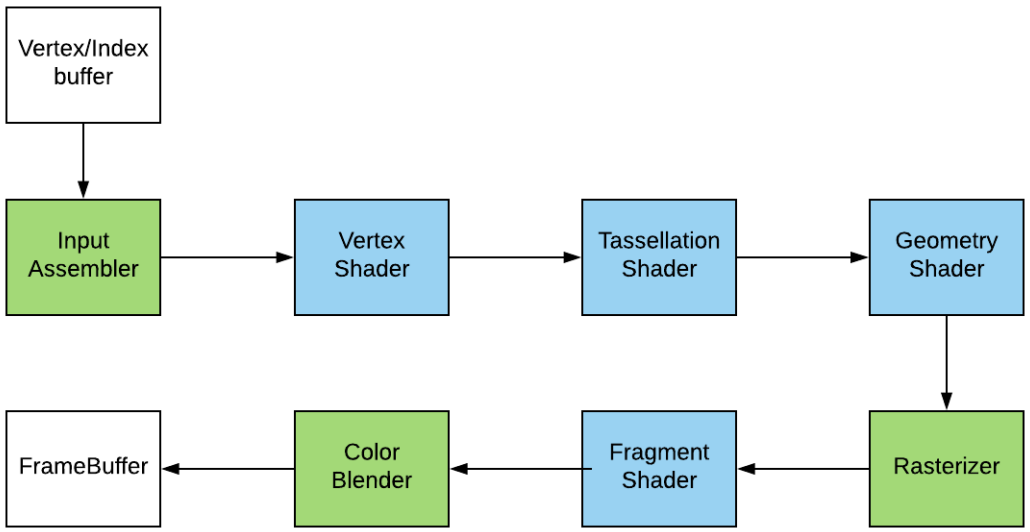

Figura 3.4: Schema della pipeline in Vulkan, in verde gli stage fissi, in azzurro quelli programmabili.

3.2.7 Descriptor Sets per gli Shader

Un'altra impostazione molto importante durante la definizione della pipeline riguarda sempre gli shader. Ogni shader oltre ai dati ricevuti per vertice(vertex shader) o per pixel(fragment shader) potrebbe anche utilizzare input tramite uniform. In Vulkan le uniform sono legali solo sotto forma di blocchi, la composizione di questi blocchi deve essere specificata a tempo di creazione e deve essere coerente con quanto dichiarato nel compilato in SPIR-V. La disposizione viene indicata fornendo un layout dei descrittori. L'oggetto *VkDescriptorSetLayout* descrive un set di di descrittori. All'interno del codice di shading, i blocchi uniform fanno riferimento ad un indice di set e un indice di binding(figura 3.5). Ogni descrittore appartiene ad un binding all'interno di un set.

Inoltre nel layout, per ogni descrittore di un set, viene specificato in quale stage della pipeline la uniform è accessibile. Per quanto riguarda i set veri e propri, questi vengono allocati a partire da una pool.

```
layout(set = 0, binding = 1) uniform uniBlock{
  mat4 P_matrix;
  mat4 V_matrix;
  Light lights[10];
  int light_count;
} uniforms;
```

Figura 3.5: Esempio di come appare un descrittore in GLSL

Vulkan utilizza oggetti come le pool per gestire sia i descriptorSet che i commandBuffer, che vedremo in seguito. Prima si inizializza una *VkDescriptorPool* specificando il numero totale sei set e dei descrittori. Non serve specificare nessun layout o dimensione dei dati. Strutture come le pool non allocano memoria, né gestiscono dati. Bensì fungono da registri in cui Vulkan segna le quantità di oggetti utilizzati. Successivamente si allocano i *VkDescriptorSet* indicando per ognuno il layout utilizzato. Per essere utili, i descrittori, cioè le uniform negli shader, devono puntare a locazioni nella memoria per fornire dati utili. I dati in memoria sono salvati attraverso buffer dati, immagini, o sampler combinati a immagine. Gli oggetti *VkBuffer* contenenti le uniforms vanno salvati nei descriptorSet con una operazione di scrittura. Successivamente per aggiornare le variabili uniform degli shader sarà sufficiente aggiornare i dati nei buffer in memoria, senza modificare i set dei descrittori.

3.2.8 Push Constant

Vulkan introduce un innovativo modo di passare dati agli shader. Le Push Constant sono indicate da uno speciale blocco negli shader, vedi figura 3.6. Queste variabili non necessitano di alcun buffer, vanno segnalate alla pipeline durante la creazione ma la loro scrittura avviene tramite esecuzione di comandi. Il comando *vkCmdPushConstants* viene incluso nei buffer di comando inviati alla coda di rendering. I dati delle push constant sono trasportati all'interno del comando. Vulkan pone un limite molto restrittivo sulle costanti di questo tipo. Il minimo supportato è 128 byte, ma nelle schede Nvidia è quasi sempre di 256 byte. Sebbene possa sembrare poco, in 128 byte ci possono stare 2 matrici di trasformazione 4x4 il che è sufficiente per l'esempio in figura 3.6. In questo caso 64 byte sono utilizzati dalla matrice di trasformazione 3D della mesh che si sta disegnando, altri 4 byte servono per salvare l'indice della texture utilizzata dal modello. L'indice

verrà utilizzato su un uniform-array di textures-combined-samplers nel fragment shader.

```
layout (push_constant) uniform PushConsts
{
  // cpu computed
  mat4 model_trasform; // 64 bytes (vec4 *4)
  // the the texture position for the current 3D mesh
  int textureIndex; // 4 bytes
} pushConsts;
```

Figura 3.6: Esempio di come appare una costante push in GLSL

Questi dati cambiano molte volte per ogni frame quanti sono gli oggetti da disegnare, senza le push constants si sarebbero dovute utilizzare delle uniform con riferimenti a buffer di memoria. Aggiornare la memoria introduce sempre delle latenze, le push constan evitano di dover effettuare trasferimenti di memoria per piccole quantità di dati che cambiano molto spesso.

3.2.9 Buffer di comando

Un'applicazione Vulkan non invia mai comandi direttamente verso una coda grafica. Invece registra in precedenza dei buffer con tutti i comandi da eseguire in ordine. In un secondo momento sottomette il buffer alla coda che lo processa appena gli è possibile. Esistono due tipi di buffer di comandi.

- Buffer di comando PRIMARI: sono indipendenti e possono essere sottomessi ad una coda di comando.
- Buffer di comando SECONDARI: sono dipendenti dai buffer primari, non possono essere inviati direttamente ad una coda, possono solo essere eseguiti da un command buffer primario.

I buffer di comando sono gli unici oggetti Vulkan che mantengono uno stato. Gli stati possibili e le loro trasizioni sono illustrati in figura 3.7.

Quando l'applicazione registra i buffer di comando, le risorse utilizzate dai comandi non vengono utilizzate. Si tratta solo di registrare un'azione futura, non si esegue nulla. Questo permette di registrare più buffer che lavorano sulle stesse risorse, in parallelo, senza preoccuparsi di eventuali conflitti. Se però si vuole fare ciò bisogna assicurarsi

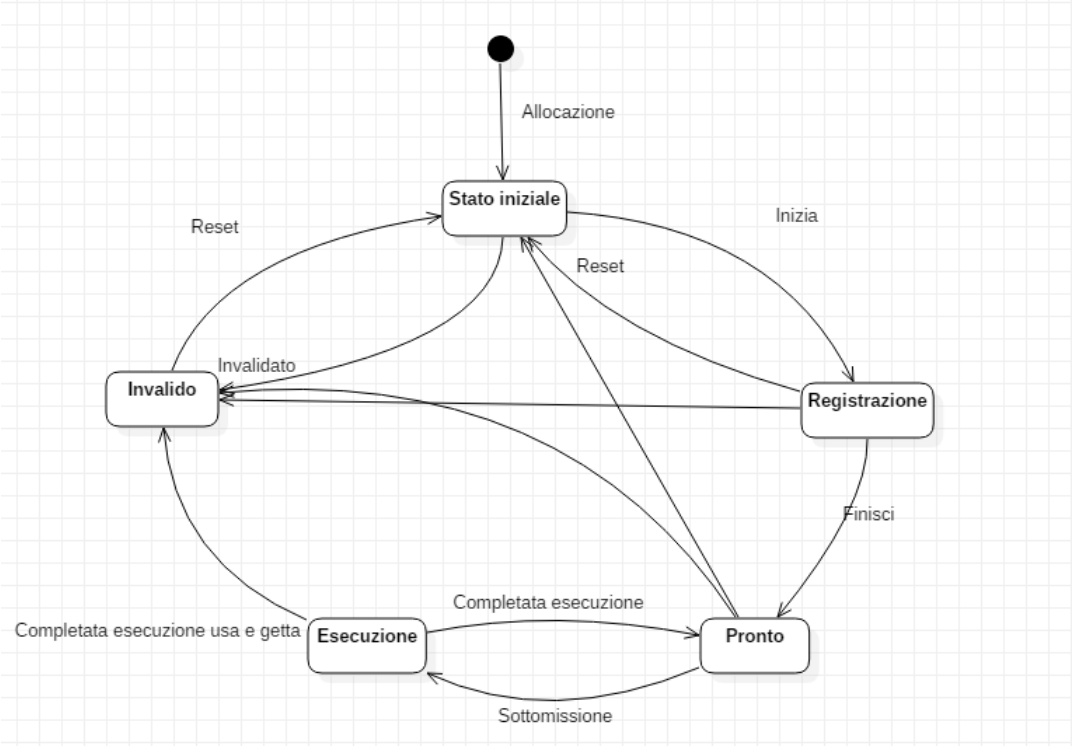

Figura 3.7: Ciclo di vita di un command buffer.

che i buffer provenienti da una stessa pool non siano registrati da più thread contemporaneamente. Ciò non comporta un problema poiché basta creare più pool, una per ogni thread. Buffer provenienti da pool diverse possono comunque essere sottomessi a fianco di altri buffer provenienti da altre pool. Questo è il concetto alla base del multithreaded rendering, implementato in questo progetto.

3.2.10 Oggetti per la Sincronizzazione del codice

Come accennato in precedenza, gestire la sincronizzazione è cruciale per un'applicazione Vulkan. Quando viene efettuata una chiamata che comporta l'esecuzione di un task da parte dell'hardware le chiamate non sono bloccanti. Le funzioni Vulkan lanciano i comandi sull'hardware e poi ritornano immediatamente, senza aspettare. Questo può portare a corse critiche quando successive chiamate utilizzano le stesse risorse.

Un esempio molto naturale è il processo di rendering, che si compone di 3 passaggi: acquisizione dell'immagine, resa della scena sull'immagine, presentazione. Questi passaggi vanno eseguiti in ordine per ogni frame della scena, scorrendo le immagini della swapchain. É ovvio che si dovrà attendere che l'acquisizione termini per poter

renderizzare. Poi solo a rendering terminato l'immagine potrà essere letta e trasferita nuovamente nella swapchain. Queste 3 azioni si compiono in 3 chiamate Vulkan, tutte dirette al dispositivo utilizzato. Vulkan ritornerà immediatamente e questo può creare enormi problemi. L'applicazione continuerebbe a sottomettere lavoro alla GPU molto più rapidamente di quanto questa riesca a smaltire. Un modo per attendere che la GPU abbia terminato il proprio lavoro è bloccare l'applicazione con la chiamata a *vkQueueWaitIdle*. Tuttavia aspettare che la GPU abbia terminato il rendering di un frame non ci permette di sottomettere più frame alla volta. Infatti sarebbe possibile compiere azioni come l'acquisizione del prossimo frame, ancora prima che la presentazione del frame precedente sia conclusa. Questo è possibile perché internamente le schede video parallelizzano tutto ciò che è parallelizzabile.

Facciamo un esempio: abbiamo in coda il comando A e poi il comando B. A e B utilizzano risorse hardware differenti. La GPU comincia ad eseguire A e senza aspettare esegue anche B. B potrebbe terminare prima di A! Ora se sostituiamo A con l'esecuzione dei buffer di comando per il rendering, lanciati con *vkQueueSubmit*, e sostituiamo B con *vkQueuePresentKHR* che ordina alla GPU di presentare l'immagine, si crea un bel problema. Vulkan offre 3 strumenti per permettere all'applicazione di gestire la sincronizzazione del codice.

- *VkSemaphore*: permette di gestire il flusso di codice all'interno della GPU. Grazie ad un semaforo, si può ordinare ad un buffer di comando di bloccarsi ed aspettare la terminazione di un buffer precedente. Questo può avvenire sia tra buffer sulla stessa coda che tra buffer su code differenti che eseguono in parallelo.

- *VkFence*: permette di bloccare il codice applicazione lato CPU, in attesa che un determinato task della GPU termini. Fondamentale per permettere al codice GPU di comunicare il suo stato alla CPU.

- *VkEvent*: permette di bloccare la GPU in attesa di un evento, che può essere generato sia all'interno della GPU(come i semafori) che dal codice CPU. La comunicazione intra-queue non è possibile, però si possono sincronizzare i singoli comandi all'interno di un singolo buffer.

In figura 3.8 è raffigurato un esempio nel quale il codice GPU è disciplinato da due semafori. In più l'applicazione aspetta di essere segnalata con una staccionata(*VkFence*) per procedere al frame successivo. Un'ottimizzazione per sfruttare la swapchain è bloccarsi su una staccionata solo quando si è già richiesto di processare tutte le immagini. In tale modo si sfrutta al meglio la configurazione triple buffering con swapchain mailbox.

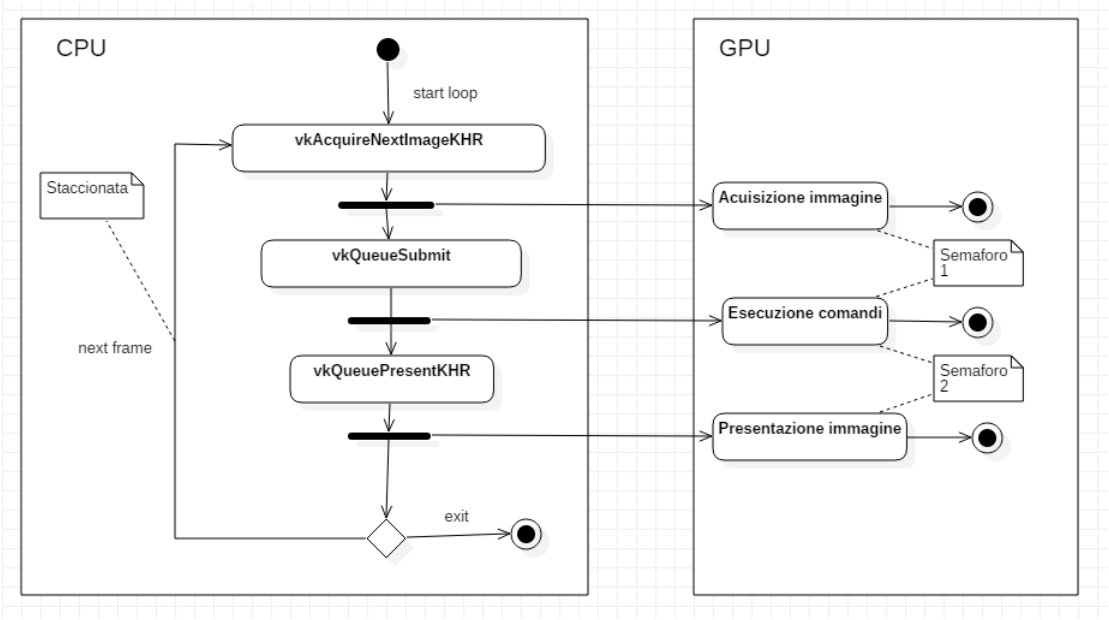

Figura 3.8: Esempio di uno scenario di sincronizzazione basilare.

3.2.11 Trasferimenti dati e Barriere di memoria

Quando vogliamo caricare dei dati in memoria dedicata, Vulkan ci costringe a definire un layout e una tipologia di heap. Il layout serve a comunicare al driver il tipo di accesso al quale la memoria verrà sottoposta. In pratica lo sviluppatore comunica l'utilizzo futuro dei dati al driver, il quale ottimizza la loro disposizione in favore del tipo di accesso che l'applicazione richiede. Lo heap invece è il tipo di memoria nel quale si vogliono salvare i bit. Le schede video possiedono diversi tipi di heap, ognuno con i relativi pro e contro. Esistono vari flag che possono descrivere uno heap. Vulkan richiede che esista almeno uno heap per ognuna delle seguenti configurazioni.

- Heap HOST_VISIBLE e HOST_COHERENT: si tratta di memoria raggiungibile dalla CPU e compatibile per scrittura e lettura.

- Heap DEVICE_LOCAL: questa memoria è locale alla GPU, la CPU non può ne scrivere ne leggere da essa.

Questi 2 tipi di memorie possono essere differenti, nel caso di una scheda dedicata, il primo è RAM, la seconda è VRAM. Se però ci si riferisce ad una scheda integrata, ad esempio una Intel serie HD, CPU e GPU sono sullo stesso chip, perciò tutta la memoria locale alla GPU sarà anche coerente e visibile all'host. Entrambi questi tipi di memoria possono essere utilizzati dalla scheda video, tuttavia l'accesso alla memoria dedicata DEVICE_LOCAL è molto più veloce che ad una memoria visibile a e coerente con la CPU. Se vogliamo utilizzare questa memoria dobbiamo effettuare dei passaggi. Di seguito è descritto il procedimento per caricare una texture sulla memoria della GPU, pronta per essere letta da uno shader. Questa tecnica è detta staging, in quanto crea un buffer solo come stage per il caricamento vero e proprio.

1. Leggere il file da disco e salvarlo in memoria.
2. Allocare memoria HOST_VISIBLE e HOST_COHERENT e creare un VkBuffer per gestirla. Questo sarà il buffer di stage.
3. Mappare una parte di memoria dell'applicazione alla memoria HOST_VISIBLE e HOST_COHERENT appena allocata.
4. Copiare i dati sulla memoria mappata. Ora il buffer è inizializzato.
5. Allocare una memoria DEVICE_LOCAL per un oggetto VkImage.
6. Trasitare il layout della memoria in un layout ottimizzato per essere destinazione di trasferimento. Si usa un command buffer.
7. Copiare l'oggetto VkBuffer nell'oggetto VkImage. Si usa un command buffer.
8. Transitare nuovamente il layout dell'immagine in uno adatto ad essere acceduto da uno shader.
9. Distruggere il VkBuffer di staging e liberare la sua memoria.

Durante questo procedimenti sono stati effettuati 2 cambi di layout. Tali cambi vengono attuati mediante l'esecuzione di particolari

oggetti chiamati *VkImageMemoryBarrier*. Questi oggetti, servono a definire transizioni di layout per la memoria di immagini Vulkan. La struttura(*VkImageMemoryBarrier*) contiene informazioni riguardo il layout di origine dell'immagine e il layout di destinazione e viene utilizzata all'interno del comando *vkCmdPipelineBarrier* per effettuare la transizione. Cosa ancora più importante, il comando vkCmdPipelineBarrier permette di specificare a quale stage della pipeline di rendering, l'immagine sarà pronta per subire la transizione, e quale stage della pipeline dovrà attendere la fine di tale transizione. Questo è il motivo per cui vengono chiamate barriere; assicurano una corretta trasformazione della memoria senza che questo alteri il corretto funzionamento della pipeline. Ovviamente lo sviluppatore deve essere parsimonioso nello specificare gli intervalli nei quali la transizione ha la priorità sull'utilizzo della memoria. Impostare barriere troppo grandi potrebbe rallentare eccessivamente l'esecuzione della GPU creando latenze superflue.

3.2.12 Panoramica

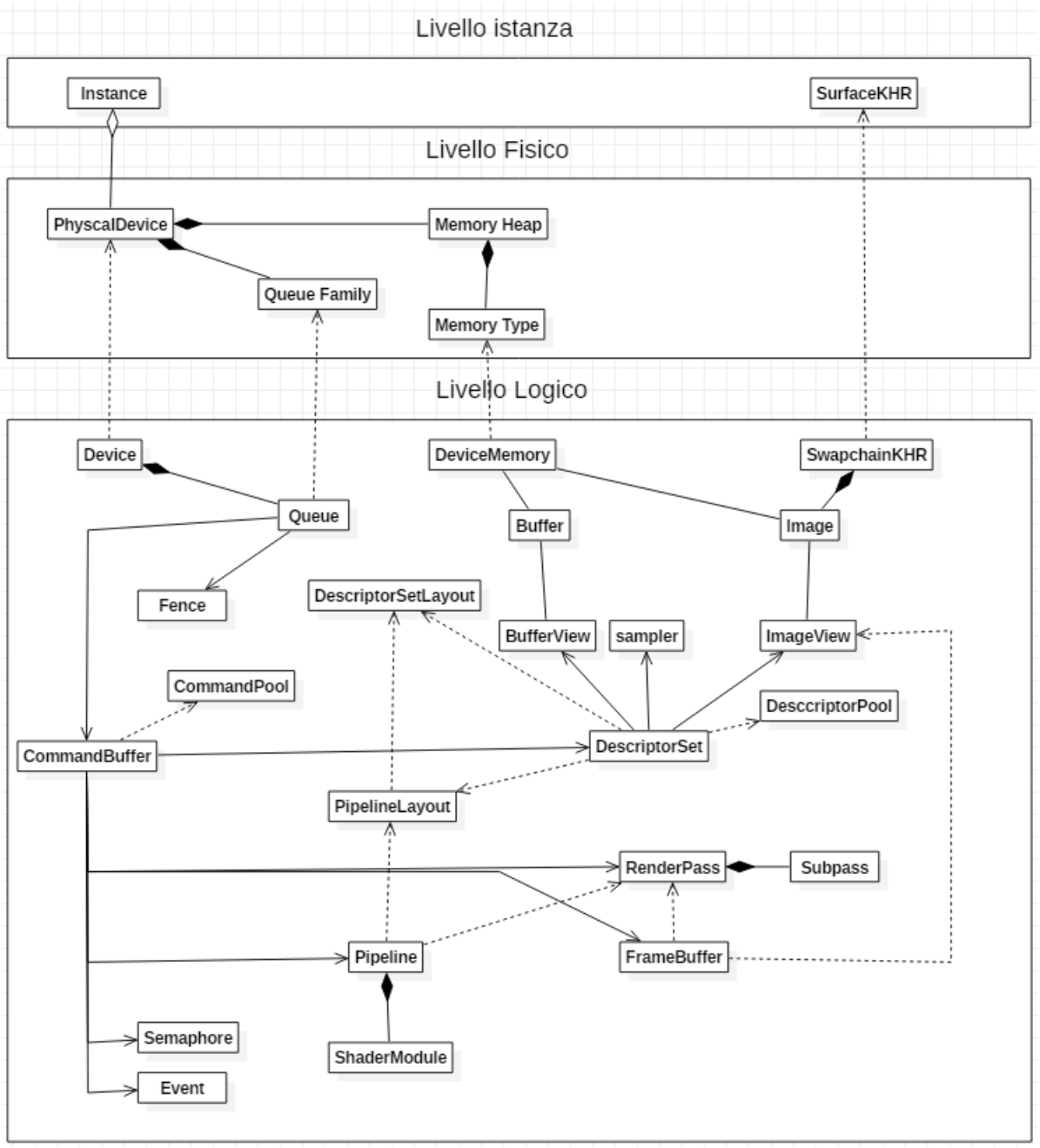

Figura 3.9: Panoramica della libreria con le principali relazioni e dipendenze ed associazioni dirette tra gli oggetti.

3.3 Estensioni, utility e tool

Vulkan viene distribuito in pacchetti di sviluppo chiamati SDK. Assieme all'API vengono forniti molti strumenti open source compilabili, tra i quali molti esempi sull'utilizzo della libreria. Il contenuto può variare a seconda del produttore della SDK. Molte soluzioni possono essere trovate anche in rete. Di seguito riporto alcuni importanti strumenti per lavorare con Vulkan.

3.3.1 Validation Layers

I livelli di validazione sono il primo strumento da adottare quando si programma con Vulkan. I livelli vengono trattati come estensioni, vanno perciò caricati sia per l'istanza che per il dispositivo logico. I livelli di validazione sono totalmente opzionali. Sono inoltre open source e possono essere scritti da chiunque. Ogni livello influisce su un sottoinsieme di funzioni di libreria, è perciò possibile attivare il debug in modo selettivo a seconda delle esigenze.

3.3.2 RenderDoc

RenderDoc rappresenta un ottimo tool di debug per applicazioni 3D. É compatibile con Windows, Linux e Android. Supporta le API: Vulkan, D3D11, D3D12, OpenGL e OpenGL ES. RenderDoc permette di lanciare applicazioni 3D e scattare speciali screenshot. Questi screenshot fotografano lo stato di esecuzione del codice all'interno della GPU. Permettono di effettuare debug sui singoli pixel dell'immagine o sul singolo vertice in uno shader. Permette di visualizzare lo stato della pipeline al momento dello screenshot. Si può inoltre visualizzare una timeline di eventi come: lettura, scrittura, ed utilizzo di barriere di memoria. RenderDoc ed altri tool di tracing funzionano grazie al vulkan Loader. Quest'ultimo permette a Renderdoc di agganciarsi al flusso per intercettare le chiamate Vulkan dell'applicazione e fare debug.

3.3.3 SPIR-V Cross

Quando solitamente si crea una pipeline in Vulkan, bisogna fornire il layout dei descrittori degli shader. Sebbene nel codice Spir-V siano già presenti le variabili di input, lo sviluppatore deve comunque definire

nel codice di inizializzazione qual'è il layout delle uniform. Questo è molto scomodo poiché se bisogna modificare lo shader, tale modifica va riportata anche nel codice applicazione.

Spir-V Cross permette di velocizzare e semplificare queste modifiche. Grazie a questa libreria è infatti possibile individuare tramite reflection il layout dei descrittori di uno shader. Così poi si potrà creare il layout per assemblare i set e impostare la pipeline, in modo automatico.

3.3.4 Vulkan Samples

Vulkan non possiede una letteratura molto sviluppata a suo supporto. Esiste ovviamente la specifica, ma una specifica non è sufficiente a chi deve comprendere la libreria. Esistono libri[1] per iniziare a lavorare e alcuni ottimi tutorial in rete. Degno di credito è il tutorial di Alexander Overvoorde [4] disponibile su `vulkan-tutorial.com`, ottimo per chi si avvicina la prima volta a Vulkan. Il passo successivo è applicare Vulkan a scenari complessi. Per avere una idea di cosa fare e come farlo, la risorse migliori rimangono gli esempi. Ogni SDK distribuisce degli esempi compilabili, inlotre alcuni esperti di Vulkan detengono utilissimi repository pubblici. I più "famosi" sono gli esempi di Sacha Willems[6], uno sviluppatore 3D che lavora presso il Khronos Group.

3.3.5 Vulkan Memory Allocator

La gestione della memoria in Vulkan può rivelarsi un compito oneroso, soprattutto quando se ne fa un uso intensivo. Le schede video hanno tutte un numero massimo di allocazioni, molto spesso si parla di 4096 allocazioni massime. Si tratta di un numero critico che non va mai superato. Quando la gestione delle allocazioni è manuale è sempre bene limitare queste ultime al minimo numero possibile. L'ideale spesso sarebbe fare una enorme allocazione in cui poi sub-allocare i buffer necessari. Implementare un allocatore è spesso un'operazione overkill per le applicazioni che non necessitano di una gestione capillare della memoria. Per questo esiste una utilissima libreria open-source, il VMA: Vulkan memory allocator. Il VMA offre al programmatore un interfacciamento per gestire la memoria Vulkan ad un livello più alto. Anche questo è un'utilissimo tool che aiuta chi vuole implementare un allocator.

3.4 Supporto

3.4.1 GPU info

Su vulkan.gpuinfo.org[3] è disponibile un database di tutto l'hardware esistente che supporti Vulkan. Sul sito sono in oltre listate tutte le features, estensioni disponibili, i formati dei dati, i tipi di memoria e gli heap che possono essere utilizzati in Vulkan. Per ognuna di queste caratteristiche, il sito fornisce statistiche su quale sia la percentuale di schede video a supportare la caratteristica. E' quindi possibile visualizzare le potenzialità e i limiti di ciascuna scheda, non che la diffusione o meno di una certa funzionalità. Questo sito è molto utile per misurare il grado di compatibilità della propria applicazione Vulkan.

Capitolo 4

Sviluppo di un motore di rendering in Vulkan

4.1 Analisi

4.1.1 Obiettivi

L'obiettivo di questo progetto è realizzare un motore di rendering utilizzando Vulkan. Il concetto di motore di rendering è spesso utilizzato per indicare prodotti software di complessità molto variabile. Fondamentalmente ciò che ci si aspetta da un motore è la possibilità di decidere come comporre una scena 3D e renderla a schermo. Oltre a ciò le soluzioni commerciali offrono anche una GUI, moduli per l'utilizzo di script, gestione delle animazioni, audio.. ecc. Prodotti di questo genere richiedono anni di lavoro ed un team di sviluppo, sono composti a moduli e si appoggiano su più API con diversi back-end. In questa sede si mira ad un prodotto semplice, con poche ma utili funzionalità, che permetta di sperimentare le nuove prestazioni della libreria Vulkan. L'idea è quella di creare un impianto di rendering ad alte prestazioni, utilizzabile per rendere facile ed immediata la composizione di scene 3D tramite codice. Si vuole creare un prodotto di natura sperimentale, qualcosa che permetta di creare demo prestazionali con agilità e semplicità.

Ottimizzazione Vulkan

Il motore realizzato implementa una tecnica di rendering che sfrutta a pieno il multithreading CPU. Per fare cio è necessario individuare quali interazioni con la libreria possono essere parallelizzate. L'obiettivo è permettere alle demo di sfruttare tutti i core e massimizzare l'utilizzo della GPU. L'idea di partenza è quella di disegnare la scena

applicando il cosiddetto frustum culling, vedi figura 4.1. Tale procedura permette di risparmiare lavoro inutile alla GPU, evitando di processare ciò che non sarebbe comunque visibile. Per fare ciò bisogna controllare per ogni mesh ad ogni frame la sua posizione e dimensione. Il renderer realizzato parallelizza il culling ed ogni altra operazione relativa ad ogni singolo oggetto 3D, comprese le interazioni con l'API per il disegno.

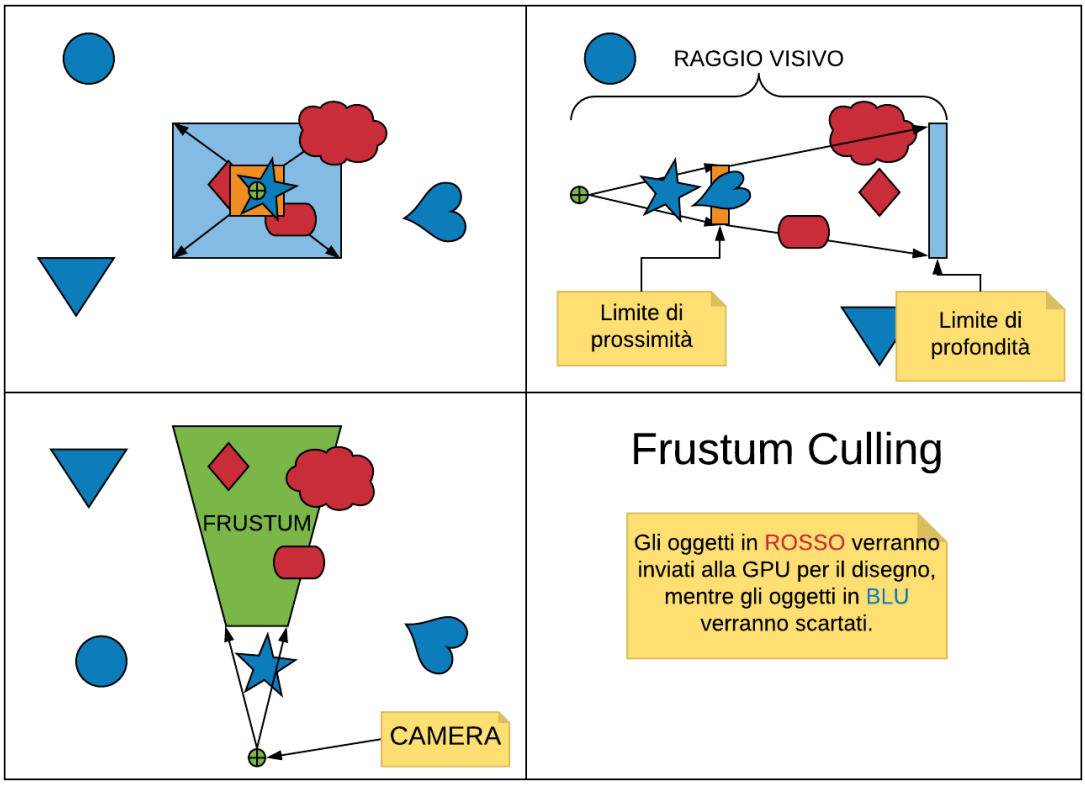

Figura 4.1: Una rappresentazione intuitiva del Frustum Culling.

Interfaccia al programmatore

Il motore permette al programmatore della demo di compiere con semplicità le seguenti azioni.

- Inizializzare il motore.
- Abilitare o disattivare i layer di validazione Vulkan.
- Caricare un numero indeterminato di modelli 3D.
- Caricare fino a 1024 file texture.
- Creare Oggetti 3D scegliendo tra le mesh caricate.

- Applicare una texture all'oggetto.
- Specificare posizione, rotazione e scalatura iniziale dell'oggetto.
- Scegliere il metodo di shading dell'oggetto.
- Inserire fino a 10 luci nella scena con colore, posizione ed intensità arbitrarie.
- Impostare varie telecamere nella scena.
- Lanciare il motore rendering.

Supporto multi-materiale

Il motore è in grado di supportare l'utilizzo di più shader. Tali shader devono essere inclusi nel motore. Il programmatore della demo può scegliere un metodo di shading diverso per ogni oggetto. Il motore offre perciò più di un materiale per la resa a schermo.

Interazione con la scena

Una volta in esecuzione, la scena 3D deve essere esplorabile nella sua interezza tramite una telecamera. Lo spostamento della camera deve avvenire lungo il piano orizzontale di vista tramite i tasti W,A,S,D. Il mouse è nascosto e bloccato sulla finestra e muovendolo è possibile ruotare in libertà la telecamera. Premendo invio la camera torna alla sua posizione di partenza.

Input utente

Una volta avviato, il motore è interrompibile tramite la pressione di un tasto. Inoltre sempre tramite tastiera, l'utente può abilitare o disabilitare le ottimizzazioni Vulkan utilizzate sulla scena. Il motore deve gestire anche il ridimensionamento della finestra.

Log a schermo

Il motore deve costantemente comunicare il frame-rate a schermo. Contare i fotogrammi prodotti ogni secondo è fondamentale per misurare le prestazioni del renderer.

4.1.2 Problematiche

Api Vulkan

Il requisito primario per la riuscita del progetto è ottenere dimestichezza con la libreria Vulkan. La libreria costringe a fare un lavoro di basso livello, con complessità maggiori esposte all'applicazione. Di conseguenza l'API richiede molte righe di codice, soprattutto nella fase di inizializzazione. L'architettura del programma deve essere in grado di organizzare il codice in maniera intelligente. Bisogna per esempio gestire in modo diverso operazioni statiche da operazioni dinamiche. Bisogna organizzare gli oggetti Vulkan secondo la loro frequenza di utilizzo e la loro gerarchia. Sarà necessario raggiungere un livello di elasticità sufficiente a soddisfare gli obiettivi di progetto. La modellazione non deve imitare troppo la struttura Vulkan, non si vuole creare una libreria contenitore, cioè un "wrapper" dell'API.

Shader Debugging

Vulkan è l'unica API a fare utilizzo dei descriptor sets, il loro uso va ad influenzare le tecniche di programmazione shader. Alcune prassi solitamente utilizzate in questi linguaggi sono incompatibili con Vulkan. Devono essere adottate nuove tecniche di interfacciamento delle uniform. Particolare attenzione deve essere data all'allineamento dati all'interno dei blocchi. Fare debug sugli shader è notevolmente più scomodo che sul normale codice applicazione. Gli shader non ritornano variabili. L'unico modo per vedere lo stack di uno shader in esecuzione è spesso mostrare particolari colori a schermo. Un'alternativa potrebbe essere utilizzare strumenti come RenderDoc.

Sincronizzazione

L' applicazione deve assumersi la responsabilità di implementare una gestione della sincronizzazione affidabile e performante. La configurazione delle guardie CPU e GPU va calibrata. L'engine deve girare in modo pulito, senza errori o leak di memoria. La sincronizzazione è vitale nel garantire la consistenza delle prestazioni durante l'esecuzione. Tuttavia una sincronizzazione fatta su tempistiche grossolane potrebbe causare blocchi non necessari del codice, compromettendo gravemente le prestazioni.

Gestione della memoria

Un altro potenziale problema potrebbe essere rappresentato dalla gestione della memoria. L'hardware grafico permette di richiedere un numero limitato di allocazioni. Se queste non dovessero bastare l'applicazione deve ricorrere a delle sub-allocazioni. Se tale necessità dovesse palesarsi, un ottimo compromesso potrebbe essere l'utilizzo del VMA: Vulkan Memory Allocator. Il tipo di demo per cui è pensato questo motore non necessita di ottimizzazioni di memoria; comunque rimane un potenziale problema che non va ignorato.

4.2 Progettazione

4.2.1 Architettura

Interfacciamento col programmatore

Anche se si tratta dell'aspetto più semplice dell'architettura, l'interfacciamento al programmatore la influenza nella sua interezza. Definendo le funzionalità a cui accede il programmatore demo se ne deriva la separazione di responsabilità tra la demo e il motore. Per questo progetto dal codice demo deve essere possibile aggiungere oggetti e luci nello spazio 3D. Prima di fare ciò è necessario richiedere al motore il caricamento di file obj e file texture, si fa tramite i metodi "loadTexture" e "loadMesh" della classe principale VkEngine. Il motore elabora questi dati, caricandoli e preparandoli all'utilizzo nella pipeline. Una volta caricati gli asset è possibile stanziare oggetti delle classi "LightSource" e "Object". Il costruttore di queste classi permette di definire trasformazioni e caratteristiche degli oggetti, incluso indice della texture e mesh precedentemente caricati. I metodi "setObjects" e "setLights" permettono di inviare agli oggetti creati al motore che li organizzerà per il rendering. In Figura 4.2 si possono osservare le classi di interfacciamento.

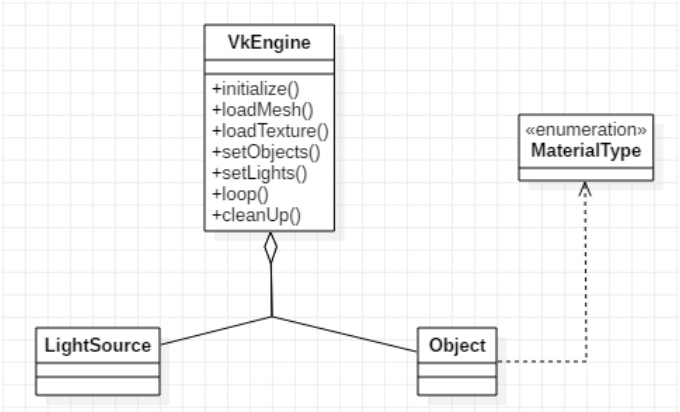

Figura 4.2: Queste sono le uniche classi e metodi che il programmatore demo deve utilizzare.

Inizializzazione del motore

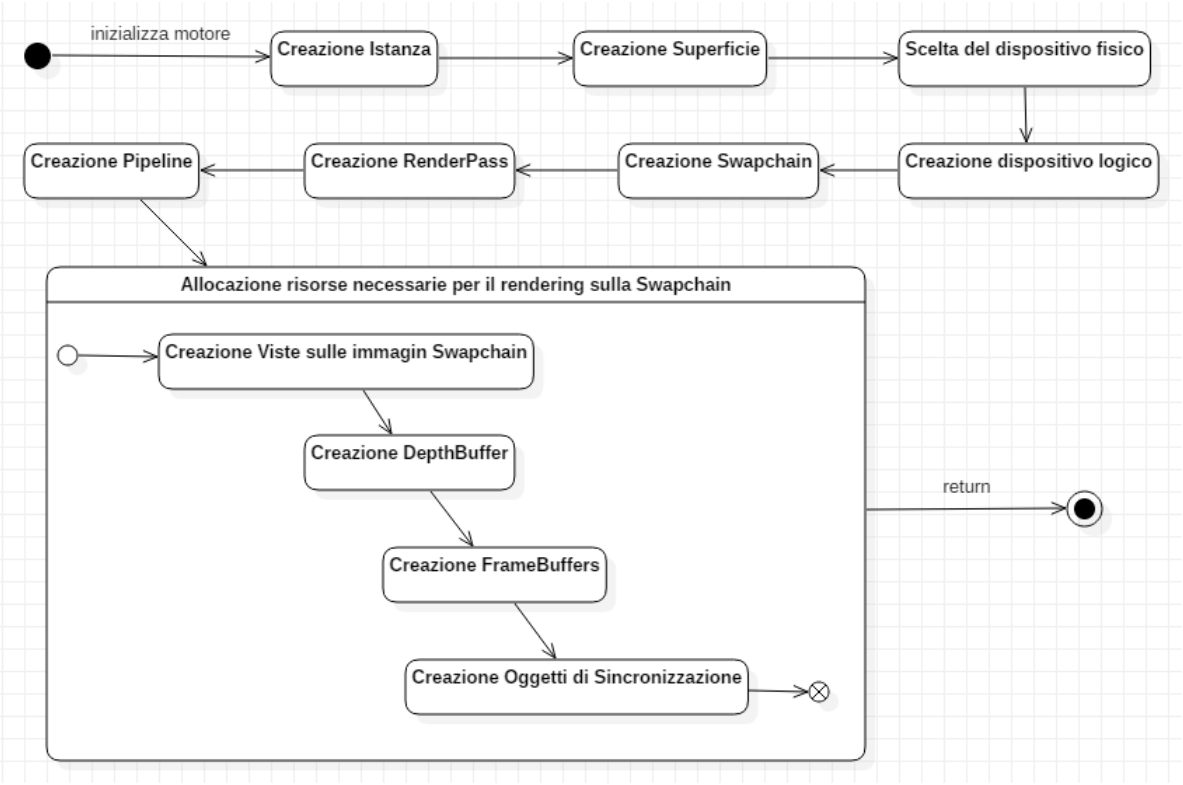

Figura 4.3: Le macro operazioni necessarie per inizializzare un motore.

In fase di analisi è stata espressa la necessità di una architettura che modellasse in modo funzionale il flusso di inizializzazione Vulkan. Tutte le applicazioni Vulkan devono seguire tale routine ma questa può modificarsi a seconda di particolari configurazioni hardware o di presentazione. L'engine di questa tesi non prevede altro utilizzo se non quello destinato al rendering su una normale finestra. D'altronde si

tratta dello scenario più comune. La figura 4.3 mostra questa routine nei suoi passaggi. Questi passaggi vanno eseguiti in fila siccome devono rispettare le dipendenze della libreria. Non è possibile risolvere questi passaggi in modo statico, ovvero con un'unico blocco funzione. Infatti quasi tutti gli oggetti Vulkan creati dovranno essere accessibile nel render loop. Persino la swapchain potrebbe cambiare. Ciò accade quando l'utente ridimensiona la finestra col mouse. É necessario perciò individuare una divisione che permetta di reiterare agilmente alcune inizializzazioni e che renda gli oggetti Vulkan accessibili ovunque essi servano.

Gli oggetti Vulkan creati durante l'inizializzazione sono singoli, cioè non ne serve mai più di uno. Perciò si è optato per una architettura a classi statiche, un compromesso tra una modellazione Object Oriented e una a funzioni. Usare la programmazione orientata agli oggetti (OOP) pura, dove ogni classe prevede la creazione di un oggetto non è sempre la soluzione migliore. Un approccio del genere risulta superfluo in alcuni scenari. In questo C++ aiuta perché permette uno stile di programmazione libero dagli schemi puri della scuola OOP. In questo progetto si è deciso di utilizzare l'OOP solo dove è più appropriato. Come si vedrà in seguito la si sfrutterà solo dove sarà necessario gestire più istanze di oggetti Vulkan.

Architettura principale a classi statiche

Riallacciandoci al discorso dell'inizializzazione, si è deciso di gestire il codice tramite classi statiche. Classi come Instance e Device devono solo essere inizializzate, le risorse che contengono sono singole e valgono per tutta l'esecuzione. Ma non si tratta unicamente di una questione di istanze delle risorse Vulkan. Il design "anti architetturale" impiegato serve a rendere di semplice accesso gli oggetti Vulkan dappertutto nel codice del motore. Va tenuto presente che la quasi totalità delle funzioni Vulkan sono funzioni legate al dispositivo. L'oggetto VkDevice contenuto nella classe statica Device deve essere referenziato ovunque nel codice. Passare gli oggetti di scope globale a tutti i costruttori avrebbe portato ad una architettura inutilmente pesante da gestire e mantenere. In figura 4.4 è possibile vedere sulla sinistra le classi ad istanza singola su cui VkEngine si appoggia, a destra invece ci sono le più importanti classi che possono essere stanziate liberamente.

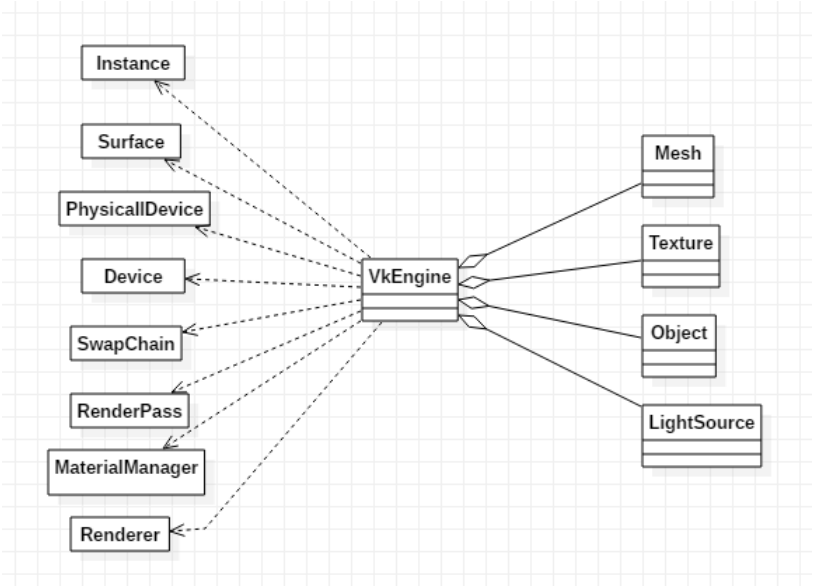

Figura 4.4: A sinistra alcune classi statiche o a singola istanza a cui è affidata l'inizializzazione degli oggetti Vulkan. A destra alcune classi usate per creare oggetti multipli.

Il sottoscritto è pienamente consapevole che una scelta di design di questo tipo non sia per nulla elegante. Tuttavia nell'ambito della creazione di questo motore sperimentale si è dimostrata vincente. Ogni classe ha le sue responsabilità e rende disponibili i propri servizi alle altre. Nella sezione sul design di dettaglio si fa luce su ognuna. Il motore inoltre si appoggia su ulteriori classi di supporto per la ricezione dell'input, la gestione dei messaggi interni, il movimento della telecamera ...ecc; tutti approfonditi nei dettagli.

Ciclo di rendering

Una volta terminata l'inizializzazione ed il caricamento delle risorse, la demo chiamerà il loop sulla classe VkEngine. Il loop raccoglie gli eventi dalla libreria GLFW, l'input grezzo viene mappato da una classe apposita e commutato in messaggio. La classe VkEngine riceve il messaggio contenente l'evento. Una volta risolti tutti gli eventi intercorsi dall'iterazione precedente, viene dato comando al renderer di disegnare il frame. Il renderer comunica all'engine se la swapchain necessita di un ridimensionamento. Questo accade se la finestra viene minimizzata o ridimentsionata. La classe principali VkEngine ha il compito di orchestrare tutte le altre componenti del programma. Il ridimensionamento della swapchain è infatti una operazione che coinvolge molti oggetti Vulkan che dalla swapchain dipendono. L'engine

dovrà distruggere e ricreare la swapchain, poi il renderpass, tutti materiali e riconfigurare il renderer. Si tratta di un'operazione molto pesante che tuttavia si innesca di rado.

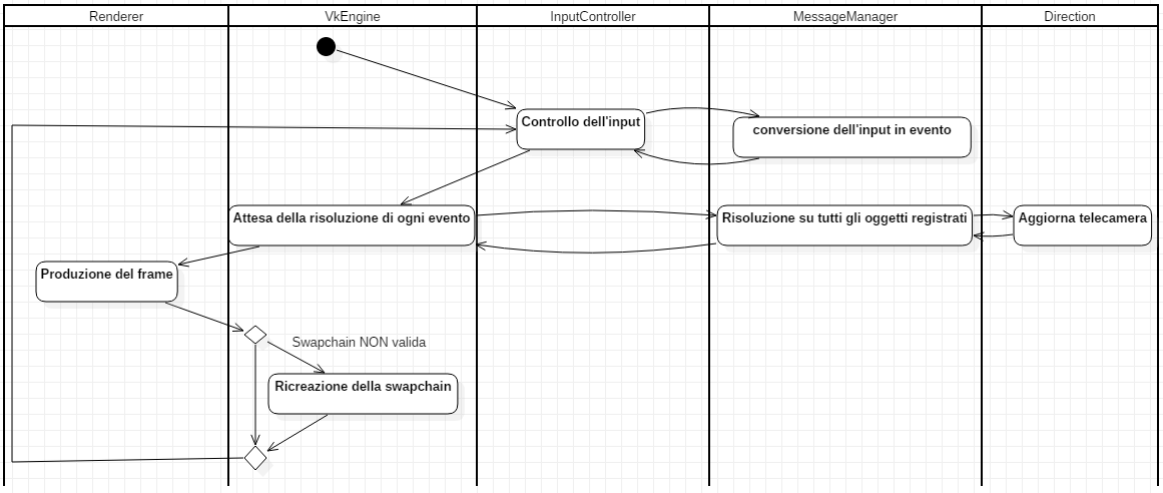

Figura 4.5: La figura mostra il flusso di controllo durante un normale utilizzo della demo.

In figura 4.5 viene mostrato il flusso di controllo tra le classi di maggiore interesse durante il render loop. Il diagramma di attività è parziale e mostra come viene distribuita la gestione degli eventi. Esiste infatti un concetto di evento di input, generato da mouse e tastiera che deve essere riconosciuto dall'InputController. Successivamente il MessageManager traduce l'input grezzo in un evento interpretabile dagli oggetti registrati. L'idea è predisporre l'engine ad una gestione eventi centralizzata ed estendibile, oltre al semplice input fisico.

4.2.2 Dettagli del Design

Di seguito è presentato un breve elenco della ripartizione di responsabilità di inizializzazione tra le classi. Queste classi sono molto dirette, non contengono logica rilevante ma prevalentemente codice di setup.

- **Instance**: si occupa della creazione di una istanza Vulkan. Permette di scegliere se abilitare o no i livelli di validazione.
- **Surface**: crea una superficie tramite l'utilizzo di una finestra GLFW.
- **PhysicalDevice**: si preoccupa di effettuare query a Vulkan sull'hardware disponibile. Effettua tutti i controlli necessari a verificare che esistano dei dispositivi capaci di soddisfare i requi-

siti minimi. Se ne individua più di uno, sceglie quello con le caratteristiche migliori.

- **Device**: crea un dispositivo logico partendo dal dispositivo fisico selezionato precedentemente. Ottiene gli indici delle code dei comandi, ed alloca le pool per i buffer di comando principlai di ogni coda.
- **SwapChain**: assembla una swapchain in base alla superficie impostando: il formato delle immagini, la loro dimensione e la modalità di presentazione. Inoltre gestisce l'acquisizione e la presentazione delle immagini durante il render loop.
- **RenderPass**: ingloba la creazione di un renderpass e dei relativi subpass.

Una volta che il motore è inizializzato è necessario caricare i dati. Il motore accetta textures e mesh in formato obj. Per gestire queste risorse sono state definite classi apposite assieme a due classi statiche per la loro gestione. Le classi MeshManager e TextureManager creano e gestiscono gli oggetti delle classi Mesh e Texture. In tal maniera il motore accede ai due manager come a due biblioteche di asset. I manager sono visibili globalmente e da essi si possono liberamente ottenere riferimenti alle texture e mesh caricate.

Gli oggetti 3D, rappresentati da istanze della classe Object, non contengono riferimenti agli oggetti Mesh o Texture, bensì mantengono solo l'indice che li identifica. L'indice viene risolto dalle classi manager, solitamente quando a tempo di rendering vengono richiesti i dati di disegno dell'oggetto. In tale modo è possibile creare infiniti oggetti che riutilizzano le stesse mesh o texture caricate. Questo è fondamentale se si vogliono gestire scene con migliaia di oggetti senza dover saturare la memoria. Infine l'oggetto contiene un codice che identifica il tipo del suo materiale. La figura 4.6 mostra la configurazione descritta.

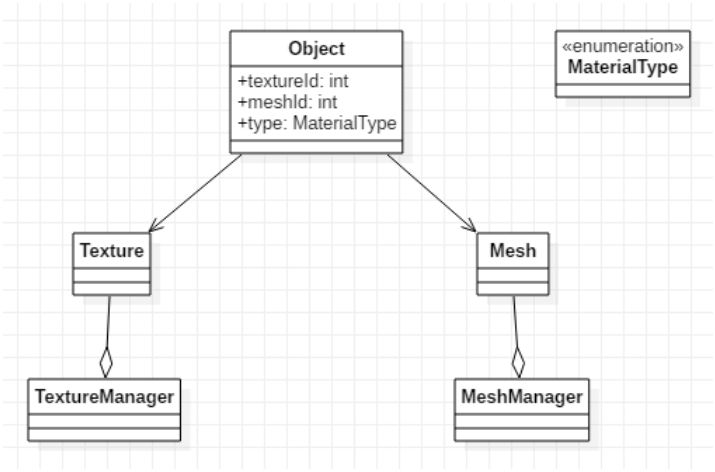

Figura 4.6: Relazioni tra un oggetto e gli asset che utilizza.

Gestione dei Materiali

Nella computer grafica a volte ci si riferisce al termine materiale intendendo concetti differenti. In un contesto di grafica fotorealistica il materiale definisce il modo in cui la superficie riflette la luce. Il più delle volte si tratta quindi di uno shader a cui vengono cambiate delle impostazioni. Invece in questo progetto si è deciso di intendere il materiale come metodo di shading. Ovvero, cambiando il materiale si cambia totalmente shader. Questo significa che è possibile avere pipeline totalmente differenti per ogni oggetto della scena. Tuttavia non è predisposta una interfaccia per la demo. Il motore contiene due diverse configurazioni con i relativi shader che il programmatore demo può selezionare. La scelta è fatta selezionando il "MateralType" per l'oggetto. Per dimostrare il funzionamento di questa feature di design, sono stati implementati due shader differenti inclusi in due materiali selezionabili. Pur fornendone solo due, il MaterialManager può essere esteso per crearne diversi. La figura 4.7 mostra le relazioni che legano l'oggetto al materiale.

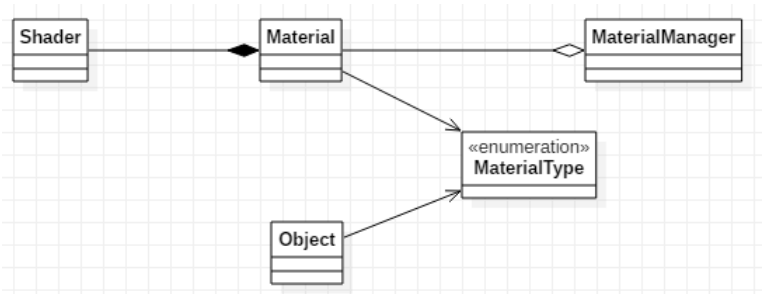

Figura 4.7: Collegamento indiretto tra oggetto e materiale.

Gestione input tramite messaggi

Come accennato in precedenza, l'engine delega la gestione dell'input ad un controller. Il controller esegue una mappatura per creare una astrazione con la libreria GLFW. Successivamente gli eventi di input grezzo sono tradotti in messaggi ed inviati agli ascoltatori. La classe MessageManager si occupa quindi della produzione dei messaggi, del loro recapito e della loro risoluzione sugli oggetti registrati.

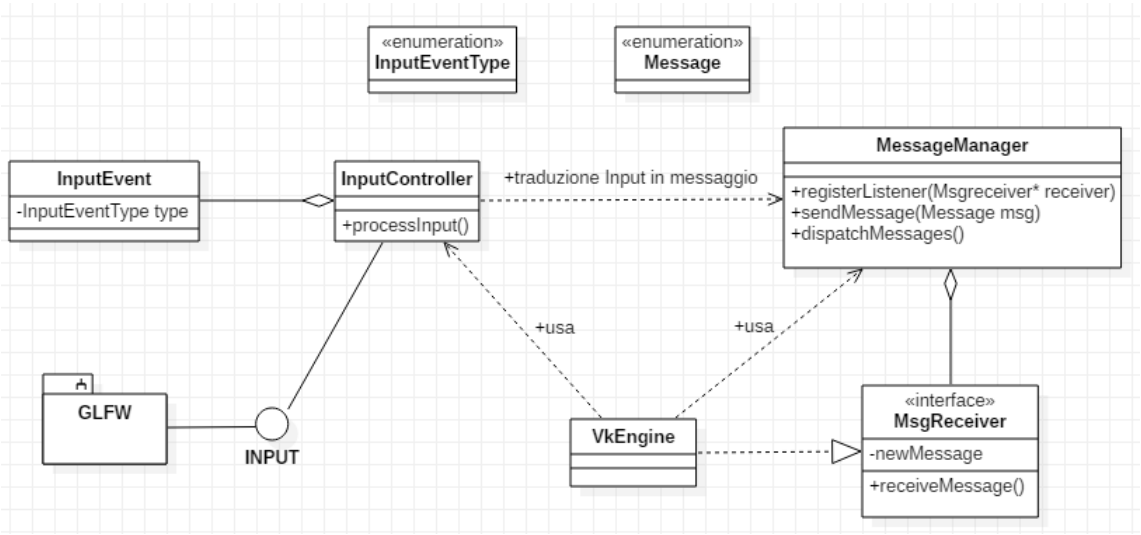

Figura 4.8: Gestione dell'input tramite eventi e messaggi.

In figura 4.8 si può notare come l'interfacciamento a GLFW è limitato all'InputContoller. Questa classe viene chiamata da GLFW per gestire gli interrupt di sistema. La classe salva gli input in eventi. Gli eventi vengono tradotti quando VkEngine richiede di processare l'input. La classe input controller traduce questo input in messaggi e li invia al MessageManager. Quest'ultimo li conserva, poi quando VkEngine ordina che i messaggi vengano risolti, il manager esegue i messaggi su tutti gli oggetti a lui registrati. Come si nota è stato utilizzato come pattern di design l'observer.

Gestione telecamere

Il motore offre una sola vista per tutta la superficie. Tuttavia si è deciso di modellare una classe Camera ed una classe Direction. L'idea è rendere semplice in futuro implementare un rendering multiview con molte telecamere nella scena. L'engine delega la gestione delle viste alla classe Direction. Questa classe statica fa da regia, gestendo gli oggetti Camera, figura 4.9.

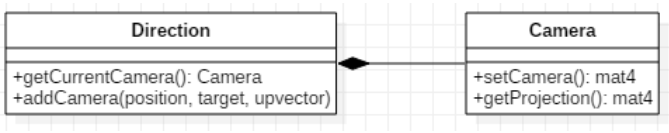

Figura 4.9: La classe direction, ritorna la telecamera correntemente selezionata.

Direction possiede anche metodi con i quali è possibile ciclare agilmente tra tutte le telecamere create. L'InputController trasmette i comandi mouse e tastiera unicamente alla camera corrente restituita da Direction. Camera è una classe con moltissimi metodi per attivare e disattivare le animazioni di movimento e rotazione. In essa sono contenuti i calcoli per generare la matrice di vista e la matrice di proiezione. É stata anche progettata per permettere in futuro la personalizzazione dei valori di proiezione. É possibile anche settare più di una modalità di navigazione. Esite una navigazione base, in cui la rotazione sull'asse verticale è bloccata sull'azimut; questo permette di muovere la telecamera come la nostra testa. Altrimenti è possibile selezionare una modalità di rotazione libera, dove l'asse verticale è sempre perpendicolare alla direzione della telecamera, ottima per scene che simulano la gravità zero.

Impianto di rendering multithread

La classe Renderer crea i frame buffer per definire il target di rendering, prepara le risorse per il rendering, gestisce la swapchain per l'acquisizione e presentazione dell'immagine e registra i buffer di comando. Per poter sfruttare Vulkan, questo progetto prevede un renderer che si appoggi su più thread. L'engine deve controllare per ogni oggetto se questo deve essere disegnato o meno i base alla sua posizione relativa al frustum. Inoltre gli oggetti 3D subiscono un'animazione di rotazione per costringere l'engine a registrare ogni frame i buffer di comando. Per ottimizzare il lavoro è necessario suddividere la composizione dei command buffer secondari su più thread possibili. Il flusso di controllo che disegna la scena è raffigurato in figura 4.10.

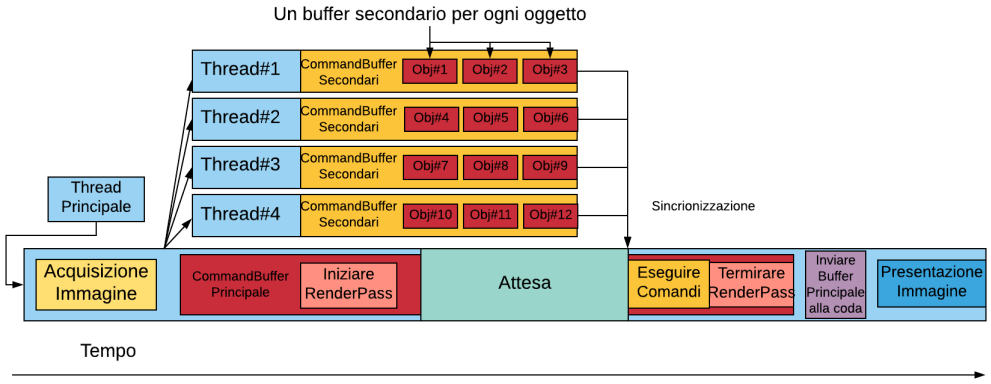

Figura 4.10: La registrazione dei comandi di disegno avviene in buffer di comando secondari, eseguiti dal buffer principale, l'architettura scala sul massimo numero di thread disponibili.

Le risorse del Renderer sono innanzitutto organizzate per thread. Ogni thread deve assemblare commandBuffers secondari che appartengano tutti alla stessa pool. Per fare questo è stata ideata la struttura ThreadData, che contiene la pool Vulkan di riferimento e tutti i command buffer secondari da essa allocati. Il numero di questi buffer è uguale al numero di oggetti 3D da disegnare. Il Renderer conta quanti oggetti deve disegnare e decide quanti command buffers ogni thread deve registrare. Poi per ogni thread alloca i buffer necessari. Il numero dei buffer però deve essere superiore. Infatti una volta registrati ed inviati alla coda di disegno, i buffer non possono essere nuovamente registrati fino alla terminazione della loro esecuzione. Per sfruttare il triple buffering con swapchain mailbox non possiamo aspettare che questi buffer terminino l'esecuzione. É necessario creare dei buffer per ogni immagine della swapchain così da permettere alla cpu di lavorare ai frame futuri senza bloccarsi ogni fotogramma. Se consideriamo B il numero dei buffer che ogni thread deve gestire, F il numero di immagini dei framebuffer,T il numero dei thread disponibili e O il numero di oggetti; la seguente formula vale.

$$B = F * O/T$$

Se con una swapchain di 3 immagini, dobbiamo disegnare 1000 oggetti e la nostra CPU ha 4 thread, sono necessarie: 4 commandPool, 750 buffer per ogni thread, 3000 buffer totali. Per rendere possibile questa ripartizione, è stata sfruttata una classe pool. La pool può essere configurata con quanti oggetti della classe Thread si vuole, figura 5.11. Ogni Thread accetta una lista di Job da eseguire. Il renderer

affida tanti job quanti sono i command buffer da registrare. Successivamente si mette in attesa sulla pool che ritorna solamente quando tutti i thread hanno terminato.

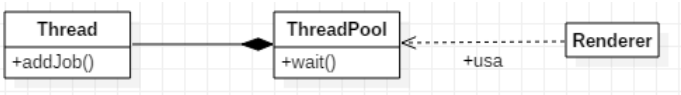

Figura 4.11: Classi a supporto del multithreading.

Progettazione della sincronizzazione

Come spiegato nel capitolo 3, Vulkan offre alcuni oggetti che ci permettono di garantire sincronizzazione all'interno del codice. La sincronizzazione è necessaria sia all'interno della produzione di un singolo frame che tra frame differenti. Questi due problemi verranno affrontati con due soluzioni diverse.

Per prima cosa è necessario assicurarsi che la GPU svolga il lavoro in modo coerente. Per quanto concerne i comandi contenuti nel buffer principale ed in quelli secondari non è necessaria alcuna sincronizzazione. La GPU eseguirà i buffer sullo stesso output di colore, il quale non deve subire trasformazioni di layout, perciò non sono necessarie barriere di memoria. La GPU eseguirà i comandi sovrapponendoli dove possibile per massimizzare la portata dati della pipeline, pur eseguendoli in ordine. La sincronizzazione ad un livello così vicino ai circuiti è gestito automaticamente. Quella che Vulkan non gestisce in automatico è la sincronizzazione tra buffer di comando separati, in questo caso se ne usa solo uno per rendere la scena, ma vale lo stesso discorso per l'acquisizione e la presentazione. Infatti il rendering non può essere completato prima di aver terminato l'acquisizione dell'immagine. Se ciò accadesse il renderpass non saprebbe dove salvare l'output di colore.

Tuttavia non è necessario aspettare il termine dell'acquisizione per iniziare il rendering. L'output di colore del renderpass viene finalizzato solo dopo che tutti i comandi di disegno hanno raggiunto lo stadio di output, oltre il fragment shader. L'ideale è lanciare il rendering senza aspettare il termine dell'acquisizione, a quest'ultima viene però agganciato un semaforo. Quando si sottometterà il buffer principale alla coda si specificheranno 2 semafori.

Il primo è il semaforo di attesa, qui è molto importante indicare lo stadio della pipeline in cui l'esecuzione deve bloccarsi. Specifican-

do lo stadio di output, quando l'acquisizione terminerà, segnalerà il semaforo e il renderpass potrà salvare l'immagine.

Il secondo semaforo svolge lo stesso lavoro ma questa volta serve alla presentazione. Questa deve avvenire solo dopo che il renderpass abbia terminato e sia avvenuta la segnalazione sul semaforo. Questo processo è illustrato dal diagramma di sequenza in figura 4.12.

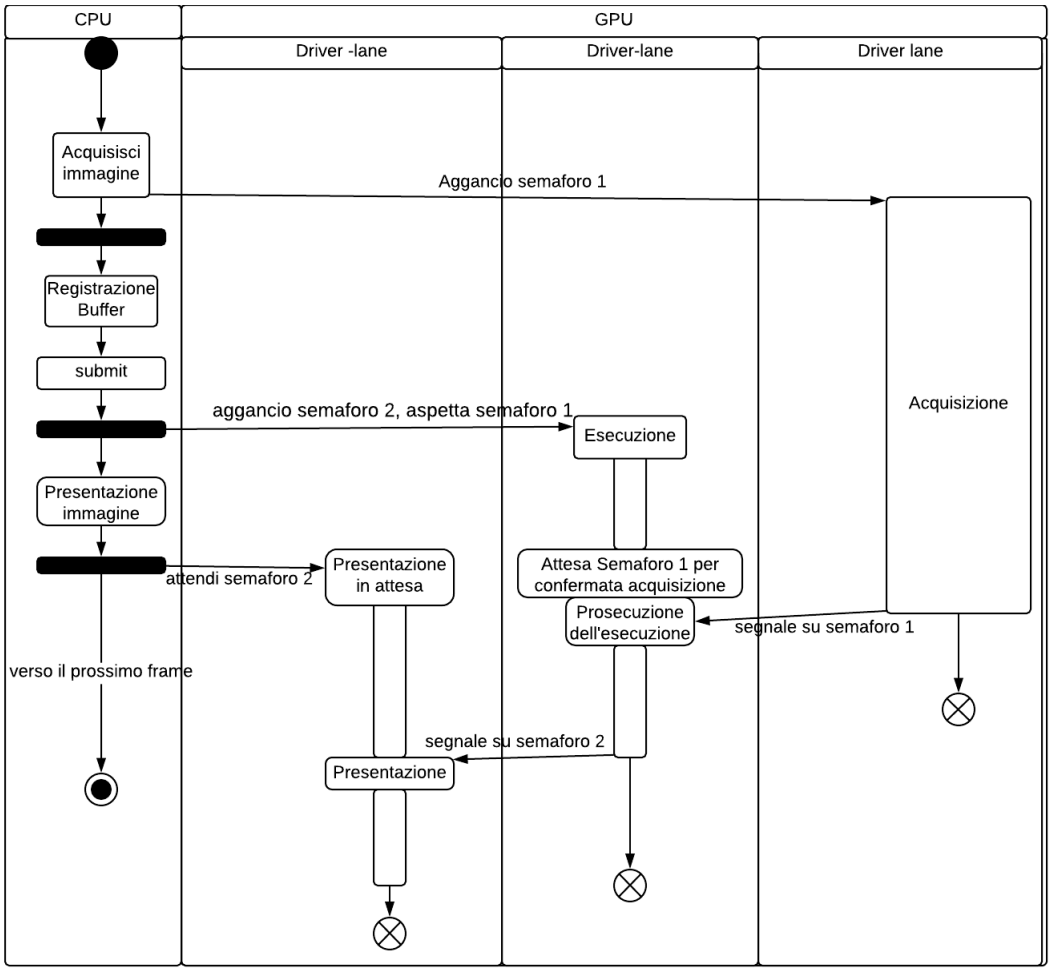

Figura 4.12: Rappresentazione dell'utilizzo dei semafori per l'output di un frame a schermo.

Questo design permette utilizzo ottimale della GPU per la produzione di un frame ma non risolve il problema per molti frame. É molto utile, per avere una immagine stabile e senza tearing, riempire sempre tutta la swapchain con immagini fresche. La swapchain del progetto è già settata per operare in triple buffering con mailbox. Per riempirla è necessario proseguire col rendering dei frame successivi senza aspertare che si concluda la terminazione di quello corrente. Ogni frame è una immagine della swapchain, i semafori sono legati al rendering su quel frame e non possono ovviamente essere usati concorrentemente

altrove. Per questo l'engine è dotato di semafori per ogni immagine della swapchain, come si è fatto per i command buffer.

Tuttavia è necessario bloccarsi per evitare di doppiare in corsa la GPU nel sottomettere frame. Per fare ciò è necessario impostare una staccionata (VkFence) per ogni renderpass dei frame. Al termine del rendering su un determinato frame, la sua staccionata verrà segnalata. In tale modo, il codice CPU potrà sapere che la resa sulla immagine ennesima è terminata e può essere nuovamente acquisita. Questo principio è ulteriormente esplicitato nel diagramma in figura 4.13.

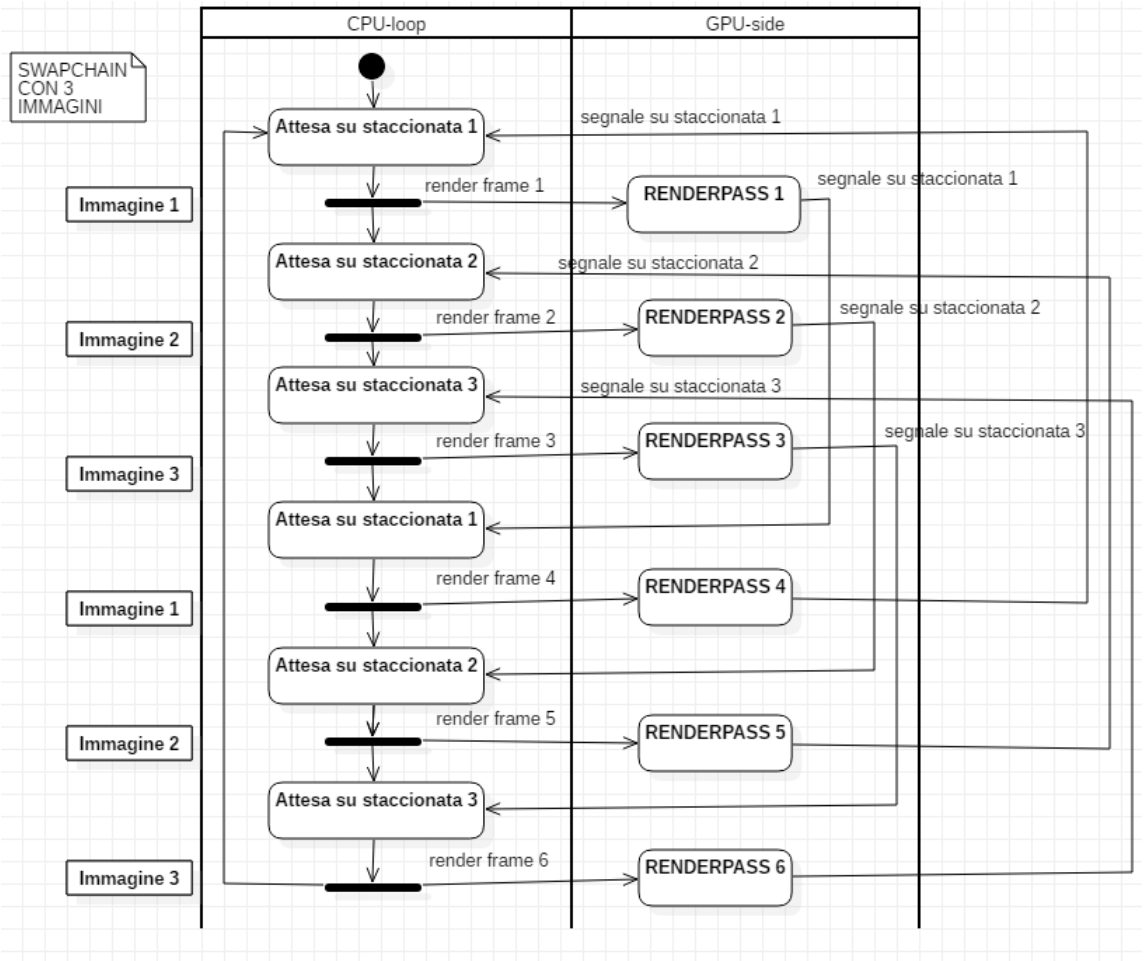

Figura 4.13: Rendering di 6 frame in loop, vengono utilizzate 3 imagini di swapchain, 1 staccionata per ogni immagine. É possibile renderizzare 3 frame in parallelo.

4.3 Implementazione

4.3.1 Ambiente di lavoro

Per sviluppare un'applicazione Vulkan è prima di tutto necessario avere a disposizione un dispositivo compatibile. L'engine è sviluppato per desktop Windows ma è compatibile anche per la compilazione su Linux. Per lo sviluppo è stato utilizzato un PC con la seguente configurazione.

- **Sistema Opertativo**: Windows 10 Pro 64-bit
- **Processore**: Intel Core™ 3770 Ivy bridge
  - **Clock**: base di 3.4 Ghz, turbo fino a 3.9 Ghz.
  - **Multithreading**: 4-core fisici , 8-threads logici.
- **Scheda Video**: Nvidia GEFORCE® GTX 1070, 8 GB GDDR5
  - **Clock**: 2000 Mhz sulla GPU, 4000 Mhz sulla memoria
  - **Memoria dedicata**: 8 GB GDDR5
- **SSD**: Samsung 970 Evo 512 GB

La compatibilità hardware non è l'unico requisito di rilievo. Questo motore dovrà essere utilizzato per programmare demo ad alte prestazioni. Quindi le caratteristiche hardware influiranno notevolmente sulle prestazioni. L'aspetto più importante in questa configurazione è la presenza di core multipli nella CPU. Gli 8 thread logici sono necessari per visualizzare i benefici del rendering multithread. Per quanto riguarda il software sono state utilizzate le risorse elencate di seguito.

- **Ambienti di sviluppo**:
  - Visual Studio 2017: programmazione C++.
  - Visual Studio Code: codifica shader GLSL.
- **Librerie C++**:
  - LunarG® Vulkan™ SDK disponibile su vulkan.lunarg.com
  - sbt:image.h disponibile su git github.com/nothings/stb/blob/master/stb_image.h per la lettura di file immagine.

  - tyny_obj_loader.h disponibile su git
    github.com/syoyo/tinyobjloader/blob/master/tiny_obj_loader.h
    per la lettura di modelli 3D in formato obj.

- **Altro software utilizzato a supporto**
  - Git: controllo di versione.
  - Blender: creazione di modelli 3D per testare il motore.

4.3.2 Metodologia

Il progetto è stato sviluppato da una singola persona. Lo sviluppo si è diviso in 3 fasi principali ognuna delle quali comprende passaggi di studio, sperimentazione e correzione.

- **Studio**: si intende la lettura di fonti cartacee come il Vulkan CookBook[1], tutorial in rete[4], articoli blog, discussioni sul forum di reddit r/vulkan[5], consultazione e studio degli esempi di Sascha Willems[6], membro del Khronos Group.
- **Sperimentazione**: tentativo di implementare il codice o le funzionalità viste in fase di studio, o di apportare modifiche al design. Consolida l'apprendimento e porta avanti il lavoro in caso di successo. Evidenzia falle nel design in caso di fallimento.
- **Correzione**: necessaria dopo ogni avanzamento. Sia che si abbia successo che meno; dopo una implementazione sperimentale viene eseguito un refactoring del codice. I refactoring ciclici permettono di mantenere il codice pulito, e nel lungo termine aiutano a prevenire i bug.

Prima fase: sperimentazione della libreria Vulkan.

Per prima cosa è stato necessario comprendere appieno il comportamento della libreria. Per questo è stato inizialmente portato avanti lo sviluppo di una demo dimostrativa. L'implementazione attuava l'inizializzazione completa della libreria ed il render loop di un triangolo. Successivamente sono state aggiunte feature base come il caricamento di texture e modelli 3D. Per ottenere questo risultato il codice è stato condensato in un'unica classe senza nessun riguardo per l'architettura.

Purtroppo in fase di studio, non essendo ancora chiaro il funzionamento generale di Vulkan, era impossibile abbozzare un design funzionale. Una volta prodotta una demo che potesse mostrare un modello 3D con texture, si poteva ottenere una visione di insieme su come utilizzare Vulkan.

Seconda fase: Implementazione del motore.

La sfida principale è stata organizzare il codice demo in maniera che fosse riutilizzabile. La demo, 1700+ righe di codice su un unico file, rappresentava un blocco monolitico contenente funzioni strettamente dipendenti da variabili globali. Una volta fatte le scelte di design discusse nella sezione precedente, si è cominciato a scomporre la demo. Questo è stato un refactoring molto lungo e impervio. Per prevenire incidenti si è usato Git come garanzia. Git permette di salvare lo storico delle versioni del codice. In tale modo, se una modifica avesse condotto ad una situazione critica, Git avrebbe conservato l'ultima versione stabile. L'implementazione del motore è quindi nata lentamente dall'implementazione della demo. Per fare ciò è stato posto come requisito di avere sempre un codice eseguibile e funzionante, ad ogni versione. Questa prassi è fondamentale, anche dopo un singolo refactoring non è ammissibile lasciare il codice in uno stato non eseguibile. Questo si applica sia a progetti sviluppati in singolo che soprattutto a progetti condivisi in team.

Terza fase: ottimizzazione del motore.

Completata la seconda fase, sono rimaste da implementare le ottimizzazione scelte per questa tesi. D'altronde, che senso avrebbe scrivere un'applicazione Vulkan, che non cerchi di trarre il maggior vantaggio dalla libreria? In fase di design è stato lungamente illustrato il concetto di rendering multithread. É stato detto come in Vulkan si possa ottenere un utilizzo parallelo dei core durante la registrazione dei buffer di comando. Per fare questo è stato molto utile consultare gli esempi si Sascha Willems[6]. Per implementare una registrazione dei buffer parallela è stato adottato un sistema, già spiegato in fase di design, in cui si allocano risorse per thread e per ogni immagine swapchain.

Un'ulteriore ottimizzazione applicata è stato il trasferimento di alcune uniform dai buffer alle push constant. Aggiornare la memoria è

sempre una operazione che introduce latenze. Per questo le scritture negli uniform buffers sono state minimizzate. Gli unici dati presenti sono relativi alla telecamera e alle luci. Questi dati vanno aggiornati una sola volta per frame. Al contrario le informazioni di trasformazione delgi oggetti e il loro indice texture cambiano di frequente durante il disegno di un singolo frame. Queste ultime variabili sono state messe dentro un blocco "push_constant" che viene inizializzato tramite dati passati all'interno di un comando. Non si scrive in memoria, ottenendo tempistiche migliori.

4.4 Dettagli implementativi

Questa sezione contiene alcuni metodi e frammenti di codice per mostrare concretamente come sono state implementate le funzionalità più importanti.

4.4.1 Utilizzo dei Validation Layers

Sin da subito sono stati utilizzati i livelli di validazione. E' impensabile programmare in Vulkan senza abilitare i livelli. Questi strumenti permettono di ricevere notifiche testuali ad alto contenuto informativo. Addirittura, per alcuni errori, viene riportato un link alla pagina inerente delle specifiche, per la consultazione. Per esempio, in figura 4.14 un livello di validazione ci notifica a riga 1 che abbiamo aggiunto un messaggero. A riga due ci dice che la funzione *VkCreateCommandBuffer* ha fallito, siccome un suo parametro era nullo. La terza riga ci dice inoltre il motivo per cui il parametro "renderpass" non può essere nullo. Per completare ci indica il capitolo della specifica che parla di come impostare i parametri per quella chiamata a funzione.

```
validation layer: Added messenger
validation layer: vkCreateFramebuffer: required parameter pCreateInfo->renderPass specified as VK_NULL_HANDLE
validation layer: Invalid RenderPass Object 0x0. The Vulkan spec states: renderPass must be a valid VkRenderPass handle (https://
www.khronos.org/registry/vulkan/specs/1.1-extensions/html/vkspec.html#VUID-VkFramebufferCreateInfo-renderPass-parameter)
```

Figura 4.14: Un esempio di un layer che intercetta una dimenticanza del programmatore.

Questo strumento è perciò utilissimo. Per poterlo usare è necessario prima di tutto caricare i layer come estensioni Vulkan, poi comunicare al Vulkan Loader quale funzione utilizzare come callback. La funzione che stampa i messaggi deve essere compatibile con l'interfaccia dei livelli di validazione. É implementata in figura 4.15.

```
1  // Funzione eseguita dalla callback
2  VKAPI_ATTR VkBool32 VKAPI_CALL debugCallbackFunction(
3    VkDebugUtilsMessageSeverityFlagBitsEXT messageSeverity,
4    VkDebugUtilsMessageTypeFlagsEXT messageType,
5    const VkDebugUtilsMessengerCallbackDataEXT* pCallbackData,
6    void* pUserData) {
7
8    std::cerr << "validation layer: " << pCallbackData->pMessage << std::endl;
9
10   return VK_FALSE;
11 }
```

Figura 4.15: Callback compatibile con i validation layer.

Per poter utilizzare la callback è necessario creare un messaggero tramite una estensione della libreria, vedere figura 4.16. Notare come nella funzione *setupDebugCallBack* si impostino i parametri per personalizzare il modo in cui i layer comunicano. Può essere impostata la severità e il tipo di messaggio. La funzione *createDebugUtilsMessengerEXT* richiede al loader l'indirizzo della funzione *vkCreateDebugUtilsMessengerEXT* per l'istanza Vulkan creata in precedenza. A questo punto a riga cinque si può finalmente agganciare la funzione in figura 4.15 e ricevere l'output dei livelli di validazione.

```
1  //Funzione "proxy" che carica la funzione "vkCreateDebugUtilsMessengerEXT" per ↩
       una istanza vulkan
2  VkResult createDebugUtilsMessengerEXT(VkInstance instance, const ↩
       VkDebugUtilsMessengerCreateInfoEXT* pCreateInfo, const VkAllocationCallbacks*↩
        pAllocator, VkDebugUtilsMessengerEXT* pCallback) {
3    auto func = (PFN_vkCreateDebugUtilsMessengerEXT)vkGetInstanceProcAddr(instance,↩
         "vkCreateDebugUtilsMessengerEXT");
4    if (func != nullptr) {
5      return func(instance, pCreateInfo, pAllocator, pCallback);
6    }
7    else {
8      return VK_ERROR_EXTENSION_NOT_PRESENT;
9    }
10 }
11
12 //Funzione "proxy" carica "vkDestroyDebugUtilsMessengerEXT" che distrugge l'↩
       estensione
13 void destroyDebugUtilsMessengerEXT(VkInstance instance, VkDebugUtilsMessengerEXT ↩
       callback, const VkAllocationCallbacks* pAllocator) {
14   auto func = (PFN_vkDestroyDebugUtilsMessengerEXT)vkGetInstanceProcAddr(instance↩
         , "vkDestroyDebugUtilsMessengerEXT");
15   if (func != nullptr) {
16     func(instance, callback, pAllocator);
17   }
18 }
19
20
21 void setupDebugCallback(VkInstance instance, VkDebugUtilsMessengerEXT* callback) ↩
       {
22   VkDebugUtilsMessengerCreateInfoEXT createInfo = {};
23   createInfo.sType = VK_STRUCTURE_TYPE_DEBUG_UTILS_MESSENGER_CREATE_INFO_EXT;
24   createInfo.messageSeverity = VK_DEBUG_UTILS_MESSAGE_SEVERITY_VERBOSE_BIT_EXT | ↩
         VK_DEBUG_UTILS_MESSAGE_SEVERITY_WARNING_BIT_EXT | ↩
         VK_DEBUG_UTILS_MESSAGE_SEVERITY_ERROR_BIT_EXT;
25   createInfo.messageType = VK_DEBUG_UTILS_MESSAGE_TYPE_GENERAL_BIT_EXT | ↩
         VK_DEBUG_UTILS_MESSAGE_TYPE_VALIDATION_BIT_EXT | ↩
         VK_DEBUG_UTILS_MESSAGE_TYPE_PERFORMANCE_BIT_EXT;
26   createInfo.pfnUserCallback = debugCallbackFunction;
27   createInfo.pUserData = nullptr; // Optional
28
29   if (createDebugUtilsMessengerEXT(instance, &createInfo, nullptr, callback) != ↩
         VK_SUCCESS) {
30     throw std::runtime_error("failed to set up debug callback!");
31   }
32 }
```

Figura 4.16: Callback compatibile con i validation layer.

4.4.2 Registrazione dei buffer di comando su più thread

Uno degli aspetti sicuramente più interessanti dell'implementazione è la realizzazione dell'impianto di registrazione multithread. Il Renderer è la classe che gestisce il disegno della scena e rappresenta il cuore pulsante del motore. Subordinata alla classe VkEngine, si occupa della preparazione ed esecuzione dei comandi di disegno. La classe prende in input i puntatori agli oggetti e alle luci della scena. Successivamente alloca le risorse necessarie a disegnare quegli oggetti. Il metodo *prepareThreadedRendering* si occupa di questo. Prima viene creato 1 buffer principale per ogni framebuffer ed una pool per buffer secondari per ogni thread; poi per ogni thread, si creano quanti command buffer

ogni thread ha da disegnare per quanti framebuffer ci sono. Il codice è mostrato in figura 4.17.

```
1  void Renderer::prepareThreadedRendering()
2  {
3    //allocazione buffer principali 1 per ogni frame
4    primaryCommandBuffers.resize(swapChainFramebuffers.size());
5    VkCommandBufferAllocateInfo allocInfo = {};
6    allocInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
7    allocInfo.commandPool = Device::getGraphicCmdPool();
8    allocInfo.level = VK_COMMAND_BUFFER_LEVEL_PRIMARY;
9    allocInfo.commandBufferCount = 1;
10
11   for (int i = 0; i< swapChainFramebuffers.size();i++) {
12     if (vkAllocateCommandBuffers(Device::get(), &allocInfo, &↩
           primaryCommandBuffers[i]) != VK_SUCCESS) {
13       throw std::runtime_error("failed to allocate primary command buffer!");
14     }
15   }
16
17
18   //allocazione pool e buffer secondari per ogni thread
19   this->per_thread_resources.resize(numThreads);
20
21   // per ogni thread
22   for (uint32_t t = 0; t < this->numThreads; t++) {
23     // 1 command pool
24     Device::createCommandPool(PhysicalDevice::getQueueFamilies().graphicsFamily,
25       &per_thread_resources[t].commandPool);
26
27     per_thread_resources[t].commandBuffers.resize(swapChainFramebuffers.size());
28
29     VkCommandBufferAllocateInfo allocInfo = {};
30     allocInfo.sType = VK_STRUCTURE_TYPE_COMMAND_BUFFER_ALLOCATE_INFO;
31     allocInfo.commandPool = per_thread_resources[t].commandPool;
32     allocInfo.level = VK_COMMAND_BUFFER_LEVEL_SECONDARY;
33     allocInfo.commandBufferCount = 1;
34     // per ogni frambuffer della scena...
35     for (uint32_t f = 0; f < swapChainFramebuffers.size(); f++) {
36       per_thread_resources[t].commandBuffers[f].resize(objXthread);
37
38       //... e per ogni oggetto da disegnare, 1 command buffer.
39       for (int i = 0; i < objXthread; i++) {
40         if (vkAllocateCommandBuffers(Device::get(), &allocInfo, &↩
               per_thread_resources[t].commandBuffers[f][i]) != VK_SUCCESS) {
41           throw std::runtime_error("failed to allocate primary command buffer!");
42         }
43       }
44     }
45   }
46 }
```

Figura 4.17: Metodo per l'allocazione dei command buffers.

La variabile "objXthread" a riga 39 viene calcolata in base ai thread disponibili secondo questa semplice logica. Se gli oggetti sono meno dei thread: il motore viaggerà su tanti thread quanti sono gli oggetti della scena. Altrimenti i thread si spartiranno gli oggetti in modo equo. Se la divisione ha un resto, questi ultimi oggetti saranno dati all'ultimo thread che avrà da disegnare una manciata di figure in più. La funzione è visibile in figura 4.18

```
1  void Renderer::findObjXthreadDivision()
2  {
3    // Trovo il numero di thread disponibili
4    numThreads = std::thread::hardware_concurrency();
5    // Se i thread superano il numero degli oggetti,
6    // riduco il numero di thread e imposto 1 oggetto per thread.
7    if (objects.size() < numThreads) {
8      numThreads = objects.size();
9      objXthread = 1;
10   }
11   else {
12     this->objXthread = objects.size() / numThreads;
13   }
14   //Imposto il numero di thread che la libreria deve utilizzare
15   thread_pool.setThreadCount(numThreads);
16   printf("\nRendering impostato su %i threads, %i oggetti ciascuno. Totale: %i ↩
         oggetti.",numThreads, objXthread,objects.size());
17 }
```

Figura 4.18: Metodo per il calcolo della ripartizione dei command buffer tra i thread.

Ogni volta che il Renderer viene chiamato a disegnare, prima di inviare i comandi alla coda di disegno, registra nuovamente tutti i command buffer relativi al frame corrente. Il Renderer durante l'inizializzazione ha creato un oggetto ThreadPool con tanti thread quanto è la variabile numThreads, vedi figura 4.19. In questo estratto si vedono i due for annidati che mettono in atto l'ottimizzazione. Questo codice si trova all'interno della registrazione di un buffer di comando principale. Ciò che fa è chiedere alla pool di aggiungere al thread di indice t il job relativo alla registrazione di un command buffer, per uno specifico framebuffer e uno specifico oggetto. Il job è rappresentato da una funzione esterna alla classe chiamata *threadrenderCode*. Questa viene passata tramite lambda. Quando eseguirà per prima cosa controllerà se l'oggetto occupa il frustum e poi registrerà i comandi per permettere il disegno dell'oggetto. Il codice prevede anche una esecuzione a thread singolo, questo per testare la differenza in prestazioni. Dopo aver lanciato tutti i thread il Renderer si blocca in attesa che finiscano i loro compiti.

```
for (uint32_t t = 0, objIndex = 0; t < numThreads; t++)
{
  for (uint32_t i = 0; i < objXthread && objIndex < objects.size(); i++, objIndex++)
  {
    VkDescriptorSet* set = DescriptorSetsFactory::getDescriptorSet(objects[objIndex]->getMatType());
    if (multithreading) {
      thread_pool.threads[t]->addJob([=] { threadRenderCode(objects[objIndex], frustum_, &per_thread_resources[t], frameBufferIndex, i, inheritanceInfo, set); });
    }
    else {
    threadRenderCode(objects[objIndex], frustum_, &per_thread_resources[t], frameBufferIndex, i, inheritanceInfo, set);
    }
  }
}
thread_pool.wait();
```

Figura 4.19: Cicli per la ramificazione in thread del renderer.

4.4.3 Shaders

In questo progetto la programmazione degli shader ha introdotto nuove sfide e possibilità. La prima è senz'altro la possibilità di utilizzare le push constant come input per variabili uniform. Il codice che segue questo paragrafo è estratto dal vertex shader, in esso si notano due blocchi. Il primo fa riferimento ad un buffer in memoria. Il buffer è individuabile tramite il layout che ne indica la locazione. Si tratta del buffer agganciato come secondo binding nel primo set.

```
layout(set = 0, binding = 1) uniform uniBlock{
  mat4 P_matrix;
  mat4 V_matrix;
  Light lights[10];
  int light_count;
} uniforms;
layout (push_constant) uniform PushConsts
{
  // cpu computed
  mat4 model_trasform; // 64 bytes (vec4 *4)
  // the texture position for the current 3D mesh
  int textureIndex; // 4 bytes
} pushConsts;
```

Il secondo blocco invece, sempre una uniform, non fa riferimento a nessun descrittore. I dati contenuti nel blocco vengono inseriti in un comando all'interno di ogni buffer secondario e sono relativi alla sua trasformazione ed alla sua texture. L'inizializzazione avviene durante la registrazione lato CPU, non avviene alcuna scrittura sulla memoria GPU. Questa è la migliore soluzione per passare dati che variano per ogni chiamata di disegno. Di seguito è presente un estratto dal codice che viaggia parallelo sui thread per la registrazione dei buffer secondari. Nel listato seguente si può notare a riga 1 che viene agganciato il

set di descrittori per gli shader. Successivamente a riga 6 si registra un comando per spingere delle costanti direttamente tramite il comando stesso.

```
1  vkCmdBindDescriptorSets(cmdBuffer, VK_PIPELINE_BIND_POINT_GRAPHICS, ↩
       pipelineLayout, 0, 1, descriptorSet, 0, nullptr);
2
3  PushConstantBlock pushConsts = {};
4  pushConsts.model_transform = obj->getMatrix();
5  pushConsts.textureIndex = obj->getTextureId();
6  vkCmdPushConstants(cmdBuffer, pipelineLayout, VK_SHADER_STAGE_VERTEX_BIT, 0, ↩
       sizeof(pushConsts), &pushConsts);
7
8  vkCmdDrawIndexed(cmdBuffer,static_cast<uint32_t>(MeshManager::getMesh(obj->↩
       getMeshId())->indices.size()), 1, 0, 0, 0);
9
10 vkEndCommandBuffer(cmdBuffer);
```

É possibile notare inoltre che a riga 6 nel comando per l'immissione delle costanti push si specifica VK_SHADER_STAGE_VERTEX_BIT. Questo significa che le costanti di push avranno come bersaglio unicamente lo shader del vertex stage. Infatti la configurazione del fragment è differete, come si puù vedere dal codice che segue.

```
1 layout(set = 0, binding = 0) uniform sampler2D texSamplers[1024];
2 layout(set = 0, binding = 1) uniform uniBlock{
3   mat4 P_matrix;
4   mat4 V_matrix;
5   Light lights[10];
6   int light_count;
7 } uniforms;
```

Come si osserva, non è presente il blocco delle costanti push, invece è presente il descrittore di binding 0 sempre del set 0. Questo è dovuto ad una scelta implementativa. Durante la creazione della pipeline si è deciso che il descrittore di binding 1 dovesse essere visibile da entrambi gli shader, siccome contiene dati uniform utili ad entrambi. Invece, il descrittore di binding 0, siccome rappresenta un array di sampler2D contenenti texture, serve solo nel fragment. Il fragment riceve l'indice per questo array dal vertex shader come input. In tale modo si minimizza l'aggancio di risorse agli shader. Questa ottimizzazione non è molto rilevante perchè il motore esegue un rendering semplice. Gli shader implementano una illuminazione che segue il modello di Phong, ma non sono molto complessi. In casi di intenso transito di uniform in sistemi di rendering complessi è utile limitare la visibilità di alcune costanti per rendere più indipendenti le esecuzioni degli shader.

Per essere in grado di interfacciarsi in questo modo con gli shader è necessario configurare correttamente la pipeline, in particolare il

layout dei descrittori.Il compito di definire quali saranno le uniform, e l'intera pipeline è della classe Material. L'istanziamento di questa classe è gestito da MaterialManager. Quest'ultimo crea tanti oggetti Material quanto è grande la enum MaterialType. Ogni MaterialType defnisce un diverso modo di inizializzarisi per la classe Material.

```
1  Material::Material(MaterialType material, SwapChain* swapchain, RenderPass* ↩
       renderPass)
2  {
3    this->swapChain = swapchain;
4    this->renderPass = renderPass;
5    switch (material)
6    {
7    case SAMPLE:
8      this->vertexShader = new Shader("Shaders/sample/vert.spv");
9      this->fragmentShader = new Shader("Shaders/sample/frag.spv");
10     break;
11   case PHONG:
12     this->vertexShader = new Shader("Shaders/phong_multi_light/vert.spv");
13     this->fragmentShader = new Shader("Shaders/phong_multi_light/frag.spv");
14     break;
15   default:
16     break;
17   }
18   createDescriptorSetLayout();
19   buildPipeline();
20 }
```

Qui sopra è riportato il costruttore della classe. Per semplicità l'unica cosa che varia sono gli shader. Anche il layout dei descrittori rimane lo stesso anche se lo shader SAMPLE non fa uso dei punti luce, siccome non utilizza un sistema di illuminazione. Il layout è importante per comunicare al driver non solo quali sono i descrittori e di che tipo, ma anche come è disposta la memoria all'interno dei blocchi. Come già accennato, bisogna porre molta attenzione siccome Vulkan non fa nulla per garantire che le strutture C++ combacino con quelle di GLSL. Di seguito viene mostrato come è definito il layout dei descrittori usati, le costanti push non sono presenti perchè non sono parte dei descriptorSet. Notare che a riga 8 viene utilizzata solo la flag per il FRAGMENT_STAGE mentre a riga 15 il secondo descrittore possiede anche il flag per il VERTEX_STAGE.

```
1  void Material::createDescriptorSetLayout()
2  {
3    VkDescriptorSetLayoutBinding samplerLayoutBinding = {};
4    samplerLayoutBinding.binding = 0;
5    samplerLayoutBinding.descriptorCount = MAX_TEXTURE_COUNT;
6    samplerLayoutBinding.descriptorType = VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER↩
         ;
7    samplerLayoutBinding.pImmutableSamplers = nullptr;
8    samplerLayoutBinding.stageFlags = VK_SHADER_STAGE_FRAGMENT_BIT;
9
10   VkDescriptorSetLayoutBinding uniformMatLayoutBinding = {};
11   uniformMatLayoutBinding.binding = 1;
```

```
12    uniformMatLayoutBinding.descriptorCount = 1;
13    uniformMatLayoutBinding.descriptorType = VK_DESCRIPTOR_TYPE_UNIFORM_BUFFER;
14    uniformMatLayoutBinding.pImmutableSamplers = nullptr;
15    uniformMatLayoutBinding.stageFlags = VK_SHADER_STAGE_VERTEX_BIT | ↩
          VK_SHADER_STAGE_FRAGMENT_BIT;
16
17    std::array<VkDescriptorSetLayoutBinding, 2> bindings = { samplerLayoutBinding, ↩
          uniformMatLayoutBinding };
18
19    VkDescriptorSetLayoutCreateInfo layoutInfo = {};
20    layoutInfo.sType = VK_STRUCTURE_TYPE_DESCRIPTOR_SET_LAYOUT_CREATE_INFO;
21    layoutInfo.bindingCount = static_cast<uint32_t>(bindings.size());
22    layoutInfo.pBindings = bindings.data();
23
24    if (vkCreateDescriptorSetLayout(Device::get(), &layoutInfo, nullptr, &↩
          descriptorSetLayout) != VK_SUCCESS) {
25      throw std::runtime_error("failed to create descriptor set layout!");
26    }
27  }
```

Per utilizzare le push costant invece, bisogna solo comunicare la struttura del loro blocco nel layout della pipeline. A riga 8 si nota come viene unicamente specificato lo shader dello stage vertex.

```
1   // Definizione del layout per caricare le uniforms sugli shaders
2   VkPipelineLayoutCreateInfo pipelineLayoutInfo = {};
3   pipelineLayoutInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
4   pipelineLayoutInfo.setLayoutCount = 1; // Optional
5   VkDescriptorSetLayout setLayout = this->descriptorSetLayout;
6   pipelineLayoutInfo.pSetLayouts = &setLayout; // Optional
7   VkPushConstantRange pushRange = {};
8   pushRange.stageFlags = VK_SHADER_STAGE_VERTEX_BIT;
9   pushRange.size = sizeof(push_obj);
10  pipelineLayoutInfo.pushConstantRangeCount = 1; // Optional
11  pipelineLayoutInfo.pPushConstantRanges = &pushRange; // Optional
12
13  if (vkCreatePipelineLayout(Device::get(), &pipelineLayoutInfo, nullptr, &↩
        pipelineLayout) != VK_SUCCESS) {
14    throw std::runtime_error("failed to create pipeline layout!");
15  }
```

Capitolo 5

BenchMarks

5.1 Obiettivo della Demo

Lo scopo per cui è stato realizzato questo pezzo di software è sperimentare le qualità di Vulkan. Nel costruirlo si è mirato al caso d'uso della creazione di una demo. É quindi il momento di mettere alla prova ciò che è stato creato. Durante lo sviluppo è stata adottata una ottimizzazione specifica, un elemento di novità, che non era fattibile con altre API. La demo deve quindi definire una circostanza che permetta al motore di sfruttare al massimo il suo vantaggio.

In pratica stiamo parlando di assemblare una scena 3D che porti al limite l'hardware. Questo può rivelarsi un aspetto non banale. Infatti, come è stato già detto in precedenza, Vulkan scala in scenari CPU-bound, cioè limitati dal processore. La demo deve quindi generare complessità nella scena in modo da sforzare parti specifiche della macchina. L'obbiettivo è costringere i thread CPU a lavorare intensamente. Allo stesso tempo è vitale tenere sotto controllo l'utilizzo GPU. É bene ricordare che la demo non deve portare l'utilizzo GPU al 100%. Se ciò accadesse il codice CPU dovrebbe rallentare poiché non può doppiare la GPU, questo è il limite imposto dalla sincronizzazione.

5.1.1 Parametri di misura

Per questa demo si è scelto di valorizzare il numero massimo di frame prodotti. L'engine è già di suo impostato per l'utilizzo di una swapchain mailbox. Significa che l'engine produrrà quanti fotogrammi possibile anche se la coda di presentazione è piena, aggiornando le immagini già incodate. Il log dell'engine permette di ricevere in tempo reale ogni secondo, quanti loop vengono eseguiti. Maggiori saranno gli FPS(fotogrammi per secondo) migliore sarà il risultato.

Altri parametri importanti sono l'utilizzo CPU e GPU. Attraverso il Task Manager di Windows 10 è possibile monitorare l'utilizzo dei thread e delle varie code di comandi della GPU. Monitorare questi valori è vitale per dare spiegazione alle variazioni di frame-rate rispetto a ciò che viene reso nella scena. Tramite l'incrocio di questi dati sarà anche possibile osservare fenomeni interessanti, come i "colli di bottiglia" all'interno di un PC.

5.1.2 Metodo di lavoro

Per ottenere delle misurazioni interessanti, i test devono essere effettuati sia in configurazione single thread che in multithread. Questo è necessario per misurare la differenza di prestazioni tra le due tecniche. Inoltre i test devono essere consistenti. L'hardware utilizzato è sempre uguale. Le componenti del PC sono le stesse riportate nel paragrafo 5.3.1 sull'ambiente di lavoro. Le prestazioni nei test sono comparate tenendo come costante la resa della stessa identica scena. Infine la stessa scena sarà anche renderizzata tramite una demo openGL per avere una diretta comparazione tra le due.

5.2 Creazione di uno scenario sintetico

5.2.1 Progettazione

L'engine una volta lanciato su qualsiasi scena spinge il computer al massimo, senza restrizioni. In base a cosa la demo decide di rappresentare si creano colli di bottiglia all'interno del PC. Per testare l'implementazione del rendering multithread è necessario fare sì che la prima componente hardware a raggiungere il massimo sforzo sia il processore. Per fare questo è necessario posizionare nella scena un grandissimo numero di oggetti 3D. Ogni oggetto viene disegnato da un command buffer che deve essere registrato su un thread. Aumentando il numero di oggetti aumenta il lavoro di registrazione a carico della CPU. Allo stesso tempo, disegnare più oggetti, causa un maggiore carico per la scheda video. Bisogna prestare attenzione a non introdurre modelli 3D troppo complessi, altrimenti il tempo di rendering diventa di gran lunga superiore al tempo di registrazione dei comandi CPU. Per questo si è scelto un modello ne troppo semplice ne troppo complesso. L'oggetto in figura 6.1 utilizza 1359 vertici, si tratta della

raffigurazione del "puzzle del millennio", tratto da una famoso cartone animato.

Il modello è stato scaricato al link www.thingiverse.com/thing:1875410. É stato creato dall'utente cinnabarsonar che lo ha gentilmente reso disponibile al pubblico per l'utilizzo sotto licenza Creative Commons, consultabile al link creativecommons.org/licenses/by/3.0/.

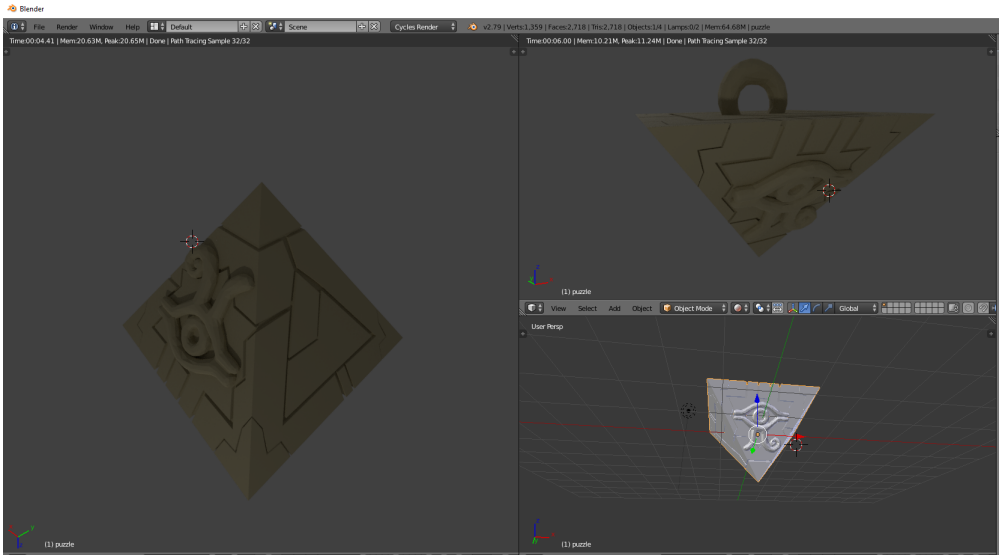

Figura 5.1: Modello 3D usato per i test, visto in Blender.

Oltre che per il numero di vertici, la scelta è puramente di gusto personale. Qualsiasi modello 3D di simile complessità andrebbe bene al fine dei test.

Per la disposizione nella scena 3D si è optato di disporre gli oggetti in modo uniforme attorno allo spettatore. L'idea è di posizionare la telecamera al centro di una matrice 3D, composta da questi oggetti ripetuti alla stessa distanza l'uno con l'altro. In questo modo si ottiene uno scenario plausibile per l'utilizzo del frustum culling. La telecamera non è in grado di vedere tutta la scena e la CPU è costretta a controllare la collisione col frustum per ogni oggetto di scena.

Vengono così aggiunti 20*20*20 oggetti alla scena e posizionati nel suddetto modo. Viene inoltre aggiunto uno sfondo tramite una grande sfera che ingloberà la scena. Siccome non è previsto alcun metodo per il caricamento di animazioni, l'engine applica di default un'animazione di rotazione. Il programmatore demo può solo specificare il vettore di rotazione e la sua velocità. La demo seleziona il materiale PHONG, reso disponibile dall'engine. Questo materiale permette di sfruttare le luci aggiunte alla scena. Il risultato visivo è reso nell'immagine 5.2.

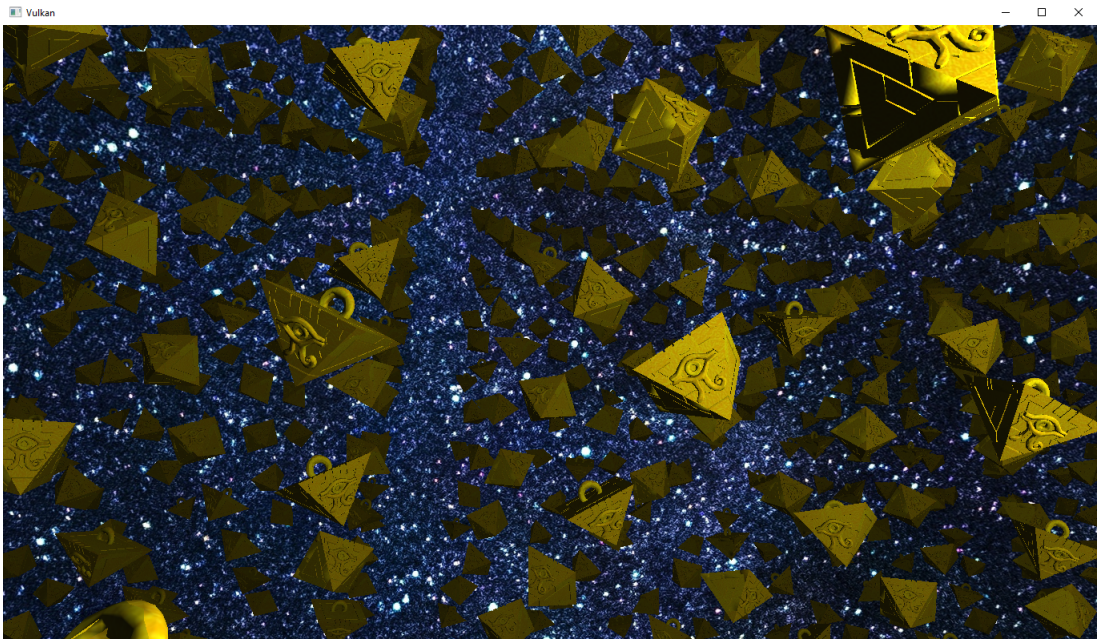

Figura 5.2: Una matrice di oggetti 3D renderizzata tramite l'engine.

5.2.2 Implementazione della demo

Ottenere questo risultato è estremamente semplice. Di seguito viene incluso il codice della demo che genera ciò che si vede in figura 6.2. Questa è la funzione main dove si inizializza l'engine.

```
1  int main() {
2    engine = new VkEngine();
3    if (enableValidationLayers) {
4      engine->validation = true;
5    }
6    try {
7      engine->initialize();
8
9      loadStuff();
10
11     engine->loop();
12
13     engine->cleanUp();
14
15     delete engine;
16   }
17   catch (const std::exception& e) {
18     std::cerr << e.what() << std::endl;
19     return EXIT_FAILURE;
20   }
21   while (getchar() != 10);
22   return EXIT_SUCCESS;
23 }
```

La funzione loadStuff a riga 10 contiene tutto il codice che carica texture e modelli; nonché la creazione degli oggetti 3D. La funzione è mostrata qui di seguito.

```
1  void loadStuff() {
2    engine->loadMesh(SKYDOME_PATH);
3    engine->loadMesh(PUZZLE_PATH);
4    engine->loadMesh(SU33_PATH);
5    engine->loadMesh(CUBE_PATH);
6    engine->loadTexture(SKYDOME_TEXTURE);
7    engine->loadTexture(PUZZLE_TEXT);
8    engine->loadTexture(SU33_TEXTURE);
9
10   Transformation transform = {};
11   transform.position = glm::vec3(0);
12   transform.rotation_vector = glm::vec3(0,1,0);
13   transform.angularSpeed = 0.0f;
14   transform.scale_vector = glm::vec3(1.0f);
15   transform.scale_factor = 100.0f;
16   Object* obj = new Object(0,MaterialType::SAMPLE,0,transform);
17   objects.push_back(obj);
18
19   int distanceMultiplier = 10;
20   int cubicExpansion = 10;
21   float rx, ry, rz;
22   for (float x = -cubicExpansion; x < cubicExpansion; x++)
23   {
24     for (float y = -cubicExpansion; y < cubicExpansion; y++)
25     {
26       for (float z = -cubicExpansion; z < cubicExpansion; z++)
27       {
28         rx = rand() % 10 * ((rand() % 2) ? -1 : 1);
29         ry = rand() % 10 * ((rand() % 2) ? -1 : 1);
30         rz = rand() % 10 * ((rand() % 2) ? -1 : 1);
31         transform.position = glm::vec3(x*distanceMultiplier, y*distanceMultiplier↩
               , z*distanceMultiplier);
32         transform.rotation_vector = glm::vec3(rx, ry, rz);
33         transform.angularSpeed = 30.0f;
34         transform.scale_vector = glm::vec3(1.0f);
35         transform.scale_factor = 1.0f;
36 \\texture e mesh sono specificati da un indice che dipende dall ordine di ↩
      caricamento
37         Object* obj = new Object(1, MaterialType::PHONG, 1, transform);
38         objects.push_back(obj);
39       }
40     }
41 ...
```

All'inizio (righe da 2 a 8) si caricano i modelli 3D e le texture. I percorsi sono stati salvati in delle costanti per comodità. Da riga 10 a riga 17 si inizializza la struttura di trasformazione che si usa subito per impostare l'oggetto 3D che sarà lo sky-dome di sfondo. I cicli for infine generano la matrice di oggetti che circonda lo spettatore. La telecamera è già di default centrata all'origine degli assi. Vengono creati 8000 oggetti. Anche se questi condividono la stessa mesh e la stessa texture, non verranno disegnati tramite instancing. Ogni oggetto è considerato unico e l'engine assume che ognuno possa avere un suo modello ed una sua texture. Successivamente si vanno a definire i punti luce che si vogliono aggiungere nella scena, qui se ne mettono nove, posizionati in modo da illuminare in modo vario la mole di figure.

```
1  ...
2    // 9 lights
3    lights.push_back(new LightSource(glm::vec3(0.0f, 0.0f, 0.0f), glm::vec3(1, 1, ↩
         1), 10.f));
```

```
4   lights.push_back(new LightSource(glm::vec3(100.0f, 100.0f, 100.0f), glm::vec3↩
        (1, 1, 1), 10.f));
5   lights.push_back(new LightSource(glm::vec3(100.0f, 100.0f, -100.0f), glm::vec3↩
        (1, 1, 1), 10.f));
6   lights.push_back(new LightSource(glm::vec3(100.0f, -100.0f, 100.0f), glm::vec3↩
        (1, 1, 1), 10.f));
7   lights.push_back(new LightSource(glm::vec3(100.0f, -100.0f, -100.0f), glm::vec3↩
        (1, 1, 1), 10.f));
8   lights.push_back(new LightSource(glm::vec3(-100.0f, 100.0f, 100.0f), glm::vec3↩
        (1, 1, 1), 10.f));
9   lights.push_back(new LightSource(glm::vec3(-100.0f, 100.0f, -100.0f), glm::vec3↩
        (1, 1, 1), 10.f));
10  lights.push_back(new LightSource(glm::vec3(-100.0f, -100.0f, 100.0f), glm::vec3↩
        (1, 1, 1), 10.f));
11  lights.push_back(new LightSource(glm::vec3(-100.0f, -100.0f, -100.0f), glm::↩
        vec3(1, 1, 1), 10.f));
12  engine->setObjects(objects);
13  engine->setLights(lights);
14 }
```

Infine si chiama "setObjects" e "setLights" per comunicare al renderer cosa dovrà disegnare. Inoltre in questo momento vengono creati i set di descrittori per l'interfacciamento con gli shader e tutte le risorse necesarie per il disegno. Ora è sufficiente chiamare il loop sull'oggetto engine e la demo processerà.

5.3 Analisi delle prestazioni dell'Engine

I test girano su una scheda video 1070 della Nvidia e su un processore i7 da 4 core e 8 thread. Con lo scenario predefinito, la telecamera è al centro e guarda fuori dalla matrice tridimensionale. Di conseguenza molti oggetti vengono scartati dal frustum e non sono disegnati dalla GPU. Di seguito il log dell'engine ci informa sul setup del rendere, poi ad ogni secondo stampa quanti fotogrammi sono stati prodotti. Come si può notare il framerate oscilla attorno ai 200 FPS. Di seguito

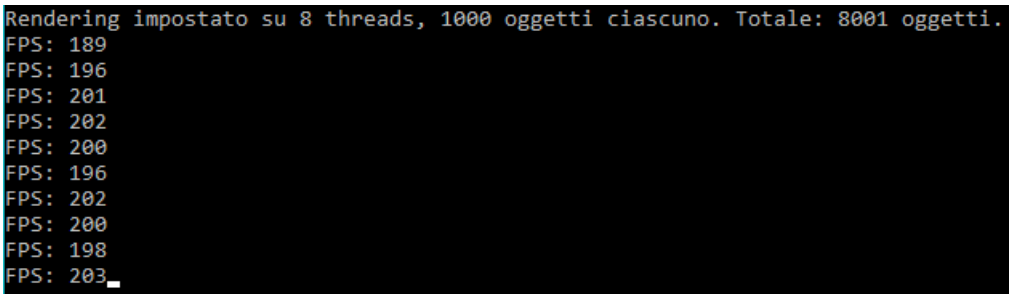

Figura 5.3: Log del primo test.

è mostrato lo stato della CPU al momento dell'avvio. Si possono notare immediatamente i benefici del multithreading. Tuttavia anche in test successivi l'utilizzo non supera mai l'80% mentre invece ci si aspetterebbe il 100%. Perché?

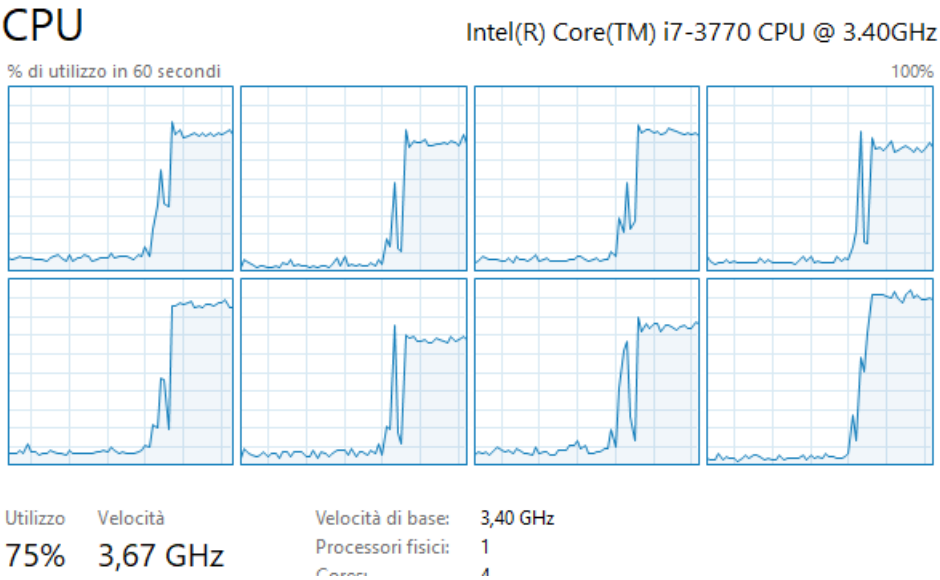

Figura 5.4: Schermata del task Manager con i thread CPU.

Non è stato indagato, ma volendo si potrebbe monitorare il flusso di codice con dei timer. Una probabile ragione potrebbe essere che il codice vada in parallelo solo durante la registrazione dei command buffers. La restante parte del loop, per quanto esigua, rimane a single thread. Inoltre bisogna conteggiare che avviene una sincronizzazione per ogni frame per sottomettere i command buffers; più una sincronizzazione eventuale ogni frame per evitare di registrare buffer ancora in stato di esecuzione.

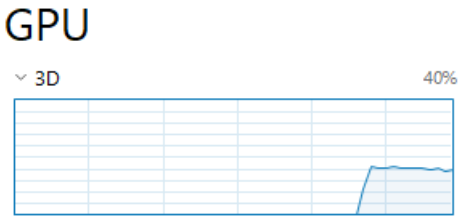

Figura 5.5: Schermata del task Manager con il grafico della GPU.

Non stupisce vedere la scheda video sotto utilizzata al 40%. Questo test, grazie al frustum culling, risparmia notevoli quantità di lavoro alla GPU. Inoltre la 1070 è una scheda di fascia medio-alta e ci vuole ben altro per saturarla.

Ora che sappiamo come performa l'engine grazie al multithreading, proviamo a disabilitarlo. L'engine ce lo permette premendo la barra spaziatrice. Ci viene immediatamente notificato il cambio di strategia e dall'output del log, possiamo vedere cosa succede, vedi figura 5.6.

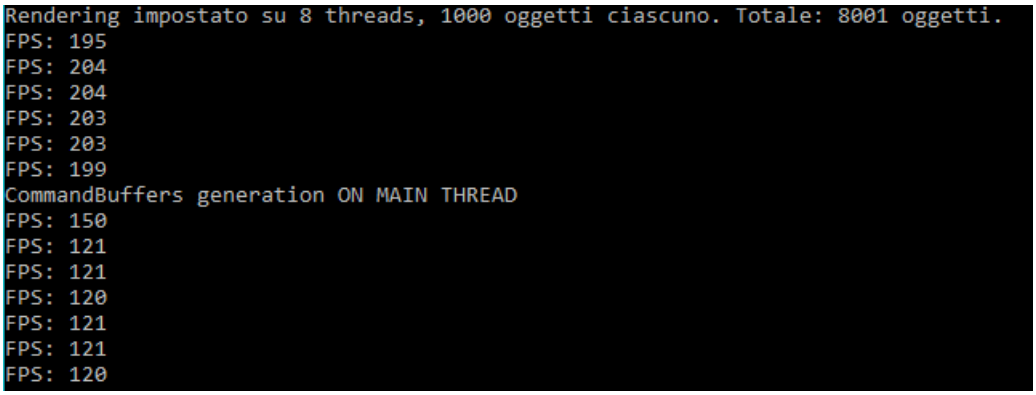

Figura 5.6: Cambio a runtime da multithread a single thread.

É evidente il drastico calo di FPS, da **200** a **120**. Contemporaneamente la CPU è calata dal **75%** di utilizzo al **22%**, chiaro segno che non avviene più alcun parallelismo. Anche la GPU ovviamente cala, dal **40%** al **25%** di utilizzo. Questi rapporti evidenziano una consistente relazione tra l'utilizzo GPU e gli FPS. A parità di geometria e shading, si possono incrementare gli FPS incrementando l'utilizzo GPU. Con Vulkan questo si ottiene parallelizzando la registrazione dei buffer. Fino a che la GPU non viene sfruttata al 100% l'unico limite rimane la CPU.

Muovendo la telecamera nella scena le prestazioni possono variare in quanto il frustum modificandosi causa variazioni nella mole di lavoro che la GPU deve sobbarcarsi. Siccome il programma consuma più CPU possibile, cercare di spostarsi verso l'esterno per inquadrare meno oggetti, non migliora le prestazioni. Questo perché il numero di oggetti da valutare è sempre lo stesso. Se invece si esce dalla massa enorme di oggetti, e la si inquadra nella sua interezza, si costringe la GPU a disegnare 8000 oggetti. Questo prima porta la 1070 al 100%, poi quando la GPU non regge più, gli FPS calano fino a 50. Una volta portata la GPU al 100% in modalità single thread, passare a multithread non porta con se alcun beneficio. Questo è il motivo per cui Vulkan è consigliato per applicazioni con importanti limitazione lato CPU.

5.4 Confronto con demo OpenGL

Per concludere i test si è deciso di ricreare la stessa scena in una demo OpenGL separata. Questa demo cerca di operare in modo ottimizzato sfruttando strutture dati moderne e senza usare vecchie funzioni

deprecate. La scena è visivamente identica, le uniche differenze sono rappresentate dalla presenza di OpenGL al posto di Vulkan.

Con grande stupore del sottoscritto, l'applicazione OpenGL è nettamente più veloce dell'engine in Vulkan. OpenGL è in grado di surclassare l'engine del **50%**. **La demo OpenGL batte la demo Vulkan in FPS: 300 a 200**. Eppure Vulkan scala sui core CPU, mentre OpenGL è bloccata e non può sfruttare il processore. La CPU infatti è bloccata al **37%** mentre invece la GPU è utilizzata al **100%**. Questo sembra essere in antitesi con quanto detto finora, o forse no.

Si è detto che Vulkan permette un maggiore utilizzo della GPU tramite la registrazione dei command buffers su thread paralleli. Questo è vero ma è anche vero che il driver Vulkan di per se non applica alcuna ottimizzazione. OpenGL invece cerca di indovinare quello che il programma disegnerà, impiega molta euristica e implementa molti stratagemmi per migliorare le prestazioni. Il risultato è che riesce a sfruttare la GPU più di quanto lo faccia il motore, nonostante il vantaggio di Vulkan. Questo engine invece implementa unicamente una ottimizzazione basata sul multithreading, e poco altro. La demo OpenGL dimostra il peso del cambio di responsabilità che si ha col driver. Implementare un sistema di disegno multithread non basta, per superare OpenGL il motore dovrebbe applicare altre tecniche.

Ma non finisce qui. Come si è detto, le chiamate alla libreria Vulkan sono meno costose di quelle ad OpenGL. Infatti se nella demo ci si sposta e si inquadrano tutti e 8000 gli oggetti notiamo un cambio di tendenza. Con la GPU al 100%, **OpenGL perde in FPS 25 a 50** contro il motore scritto con Vulkan. Questo potrebbe avvenire siccome il motore Vulkan ha impostato la pipeline in modo preciso sin dalla inizializzazione e viaggia senza controllo di errore. OpenGL invece mantiene un overhead maggiore. Quando poi il numero di chiamate di disegno aumenta notevolmente l'overhead diventa importante.

Va in ultima analisi considerato che il motore registra nuovamente a ogni frame tutti i comandi per il disegno degli oggetti dentro il frustum. Questo avviene perché le informazioni di trasformazione vengono passate attraverso le push constant. La demo OpenGL probabilmente(non si può essere certi), al suo interno salva le chiamate all'hardware senza ridefinirle.

Capitolo 6

Conclusioni e sviluppi futuri

Questo progetto ha messo in luce vantaggi e svantaggi nell'utilizzo della libreria Vulkan. Vulkan offre grandiose possibilità per l'industria, ma richiede grandi capacità e molto lavoro. La libreria è tuttora una novità e non esistono tecniche consolidate per il suo utilizzo. Sviluppare questo motore si è rivelata essere una sfida, una sperimentazione. Il motore è un successo sotto vari punti di vista, sebbene il confronto con OpenGL dimostri che rimanga ancora molto lavoro da fare. L'implementazione di un sistema di rendering multithread è un fattore di innovazione e un caso di studio molto interessante. Questa tesi scalfisce la superficie, aprendo la strada a mille altre sperimentazioni con l'API. Sebbene non rappresenti un prodotto vendibile, questo motore pone le basi per sviluppi futuri. Non solo rimane un banco di prova per migliorare l'utilizzo dell'API e ampliare le conoscenze in materia, ma offre anche nuove prospettive.

Pezzi di software come i motori di rendering vengono utilizzati ovunque, in tutte le applicazioni grafiche. Lo studio di strumenti tanto complessi è il primo passo verso l'implementazione di veri e propri framework. Basti pensare a quale mole di risorse, un videogioco necessiti per poter essere sviluppato. Il suo motore di rendering deve essere molto complesso per poter gestire una vasta gamma di scenari. Inoltre deve essere organizzato come modulo. Infatti il motore di rendering è solo un tassello al quale sono agganciati altri elementi, come per esempio un motore fisico o un editor con una GUI. Sistemi software di questo genere necessitano un alto grado di complessità per poter gestire una grande varietà di risorse.

Il prossimo passo è senz'altro studiare come migliorare ulteriormente le prestazioni. Non solo introducendo nuove ottimizzazioni ma anche creando algoritmi che individuino le migliori strategie di basso

livello per gestire gli asset che devono essere utilizzati. Il passo successivo è studiare le architetture avanzate dei sistemi grafici messi in opera dalle grandi aziende. Sarà certamente necessario adottare architetture a moduli e multi livello. I motori commerciali per esempio offrono tutti un renderer di alto livello che si interfaccia a vari backend rappresentati da motori di basso livello che a loro volta utilizzano API grafiche.

É intenzione del sottoscritto proseguire su questa strada per accumulare più conoscenza possibile in materia. Essere in grado di padroneggiare le più avanzate tecnologie di rendering, permette di sfruttare l'hardware per dare vita e bellezza ai medium più complessi e coinvolgenti che esistano, i videogiochi.

Ringraziamenti

Vorrei qui ringraziare la Dott.ssa Damiana Lazzaro per avermi permesso di realizzare questa tesi. Un grazie per avermi sempre sostenuto e aver sempre creduto nel valore dei miei progetti. Grazie anche per la disponibilità e gentilezza che ha sempre riservato a me ed ai suoi studenti. Un grande grazie infine per avermi insegnato le basi della computer grafica, una scienza che amo e della quale non voglio mai smettere di imparare.

Un grazie infine va alla mia famiglia che mi ha permesso di concentrare il mio tempo e le mie energie in questo progetto e che mi sostiene sempre nella vita.

Bibliografia

[1] Pavel Lapinsky. *Vulkan Cookbook*. Packt Publishing, Birmingham - Mumbai, 2017.

[2] Khronos Group, *Vulkan 1.1 API Specifications*. www.khronos.org/registry/vulkan/

[3] Sascha Willems, *Vulkan Hardware Database*. vulkan.gpuinfo.org

[4] Alexander Overvoorde, *Vulkan tutorial*. vulkan-tutorial.com/Introduction

[5] Community di reddit *r/vulkan*. www.reddit.com/r/vulkan/

[6] Sascha Willems, *Vulkan Examples* github.com/SaschaWillems/Vulkan