

SUPSI

XR Bridge: un wrapper OpenXR-OpenGL per applicazioni VR

Studente/i

Lorenzo Adam Piazza

Relatore

Peternier Achille

Correlatore

-

Committente

Peternier Achille

Corso di laurea

**Ingegneria informatica
(Informatica TP)**

Modulo

C10826

Anno

2023 / 2024

Data

23 agosto 2024

Indice

1	Introduzione	1
1.1	Requisiti	2
2	Stato dell'arte	3
2.1	Game engines	3
2.2	OpenVR	4
2.3	OpenXR	4
2.4	SteamVR	4
2.5	OvVR	4
3	Design e implementazione	5
3.1	API	5
3.1.1	Inizializzazione	6
3.1.2	Update	6
3.1.3	Render	7
3.1.4	De-inizializzazione	9
3.1.5	Gestione errori	9
3.2	Strumenti e linguaggi di programmazione	10
3.3	OpenXR	10
3.3.1	Documentazione	10
3.3.2	Versioni	11
3.3.3	Runtime	11
3.3.4	Loader	11
3.3.5	Filosofia della API	12
3.3.6	Layers e estensioni	12
3.3.7	Binding grafico	13
3.3.7.1	Binding OpenGL + Windows	13
3.3.7.2	Binding OpenGL + Linux	14
3.3.8	Swapchain	15
3.3.9	Spazio di riferimento	16

3.4	Limitazioni di FreeGLUT	16
3.5	Differenze fra piattaforme Windows e Linux	18
4	Risultati	19
5	Conclusioni	21
5.0.1	Applicazione di test	21
5.1	Sezione	21

Elenco delle figure

3.1	Flusso di esecuzione di XrBridge	5
-----	--	---

Elenco delle tabelle

Capitolo 1

Introduzione

Lo scopo del progetto è sviluppare un wrapper (d'ora in poi chiamato XrBridge) attorno a OpenXR per permettere di sviluppare applicazioni che fanno uso di realtà virtuale e OpenGL in modo più semplice. XrBridge andrà a sostituire OvVR, un'implementazione simile già esistente sviluppata dal docente responsabile che fa uso di OpenVR. Dal momento che XrBridge verrà utilizzato nel corso di realtà virtuale (successore del corso di grafica), esso dovrà essere il più simile a OvVR possibile.

Aver personalmente seguito sia il corso obbligatorio di grafica e il corso opzionale di realtà virtuale mi ha permesso di meglio capire i requisiti del progetto, sia dal punto di vista del docente che dovrà lavorare con XrBridge durante il suo corso, sia dal punto di vista dello studente che dovrà sviluppare un progetto che verrà poi valutato facendo uso di XrBridge.

L'aspetto più importante di XrBridge (oltre al fatto che deve funzionare) è la semplicità. Una soluzione troppo complessa sarebbe un problema sia per il docente, sia per lo studente. Una soluzione troppo complessa forzerebbe il docente a dedicare più tempo a spiegare agli studenti come funziona e come si utilizza XrBridge; il corso, infatti, è dedicato alla realtà virtuale, non a XrBridge. Tale soluzione causerebbe troppe difficoltà per gli studenti, poiché aggiungerebbe ancora più materiale da studiare e comprendere. Per sviluppare XrBridge è dunque molto importante sempre considerare il contesto in cui esso verrà utilizzato.

Lo scopo del corso di realtà virtuale è sviluppare un'applicazione in C++ che fa uso di realtà virtuale senza utilizzare game engines interagendo direttamente con API di basso livello come OpenGL e OpenVR. Il corso opzionale di realtà virtuale si basa sul corso obbligatorio di grafica, dove lo scopo è sviluppare un'applicazione grafica 3D (ad esempio un piccolo gioco) senza fare affidamento a game engines esistenti o strumenti simili. Gli studenti imparano ad utilizzare OpenGL e a sviluppare personalmente un game engine che dovranno poi utilizzare per sviluppare l'applicazione grafica. Il corso di realtà virtuale consiste, in poche parole, ad estendere l'applicazione grafica sviluppata nel corso di grafica per aggiungere la funzionalità di realtà virtuale.

1.1 Requisiti

ID	Descrizione	Note
R-01	XrBridge deve supportare la API grafica OpenGL	-
R-02	XrBridge deve supportare la piattaforma Linux	-
R-03	XrBridge deve supportare la piattaforma Windows	-
R-04	XrBridge deve supportare la runtime SteamVR	-
R-05	XrBridge deve supportare OpenXR versione 1.1	-
R-06	La API di XrBridge deve essere chiara e semplice	Il più simile possibile a OvVR.
R-07	XrBridge deve essere utilizzabile con C++	Il corso di realtà virtuale si svolge utilizzando C++.
R-08	XrBridge deve supportare head-mounted-displays	Visori di realtà virtuale da indossare oppure uno smartphone. Durante il corso di realtà virtuale si usa lo smartphone.
R-09	La API deve permettere all'utente di reperire posizione e rotazione del visore	-
R-10	La API deve permettere di rilevare l'input dall'utente	-
R-11	XrBridge deve supportare la libreria grafica FreeGLUT	L'uso di FreeGLUT è richiesto dai corsi di grafica e realtà virtuale.

Capitolo 2

Stato dell'arte

Grazie all'evoluzione della tecnologia hardware e software nel corso degli ultimi decenni, è diventato sempre più facile sviluppare applicazioni di realtà virtuale di qualità sempre maggiore e sempre più immersive. È facile riconoscere questo progresso; basta confrontare le prime esperienze di realtà virtuale, come il Nintendo Virtual Boy, e confrontarle con i videogiochi VR di ultima generazione. Nel corso del tempo sono nati una moltitudine di strumenti per facilitare lo sviluppo di applicazioni VR, da hardware facilmente accessibile a utenti casalinghi a programmi che permettono di creare scenari immersivi trascinando con il mouse oggetti in una scena virtuale.

Ci sono diversi metodi e strumenti per sviluppare applicazioni VR; di seguito ne elencherò alcuni e descriverò in che modo XrBridge si differenzierà dalle soluzioni già esistenti. Alcuni di questi strumenti sono già utilizzati nel corso di realtà virtuale.

2.1 Game engines

Nella maggior parte dei casi, appoggiarsi su di un game engine già esistente è la scelta migliore, semplice e diretta per sviluppare un'applicazione di realtà virtuale. Un game engine permette di concentrarsi interamente sul contenuto dell'applicazione senza dover pensare a tutti i dettagli necessari per avere un'applicazione grafica funzionante. Alcuni dei game engines più conosciuti che supportano lo sviluppo di applicazioni VR sono Unity, Unreal Engine e Godot.

Ci sono però alcuni scenari dove potrebbe essere necessario sviluppare un engine da zero; è il caso di un'applicazione con bisogni molto specifici non coperti da un engine già esistenti oppure, come in questo caso, se l'obiettivo è imparare a sviluppare un'applicazione partendo dalle basi. In questi casi, è necessario imparare ad utilizzarli strumenti a livelli più bassi; le prossime sezioni sono dedicate ad alcuni di questi strumenti.

2.2 OpenVR

OpenVR si tratta di un SDK e API sviluppati da Valve per facilitare lo sviluppo di applicazioni VR. OpenVR è progettato per essere semplice da usare e si concentra principalmente su applicazioni per Head Mounted Displays (classici visori a occhiali). Per questo motivo, OpenVR è più limitato a confronto con OpenXR.

2.3 OpenXR

OpenXR si tratta di uno standard aperto creato da Khronos (lo stesso gruppo che ha sviluppato OpenGL e Vulkan) con lo scopo di sviluppare applicazioni che fanno uso di realtà virtuale e realtà aumentata. La prima versione completa (versione 1.0) è stata rilasciata nel 2019 con lo scopo di risolvere la frammentazione che esiste attualmente nel mondo della realtà virtuale. A differenza di OpenVR, OpenXR è solamente uno standard che descrive una API e non offre nessun software già pronto ed è compito di produttori di hardware o piattaforme di sviluppare i software che implementano la API. Questi software vengono chiamati *runtime*. Il vantaggio di avere una API standard è quello di permettere di sviluppare applicazioni che possono essere eseguite su una moltitudine di dispositivi. OpenXR è progettato per supportare moltissimi possibili scenari, dalla realtà aumentata, alla realtà virtuale utilizzando un headset ad un sistema cave. Questa flessibilità però viene al prezzo di una maggiore complessità rispetto a OpenVR.

2.4 SteamVR

SteamVR si tratta di un software sviluppato da Valve e distribuito attraverso la piattaforma Steam. SteamVR supporta Windows e Linux. SteamVR funge sia da implementazione di OpenVR, sia come runtime di OpenXR. Tutti i videogiochi che fanno uso di realtà virtuale distribuiti attraverso Steam fanno uso di SteamVR.

2.5 OvVR

OvVR è una libreria che funge da wrapper attorno a OpenVR sviluppata dal docente responsabile ed ha lo scopo di semplificare lo sviluppo di applicazioni VR per il corso di realtà virtuale. XrBridge andrà a sostituire questa libreria. OvVR è scritto in C++ ed è interamente contenuto in un singolo file header.

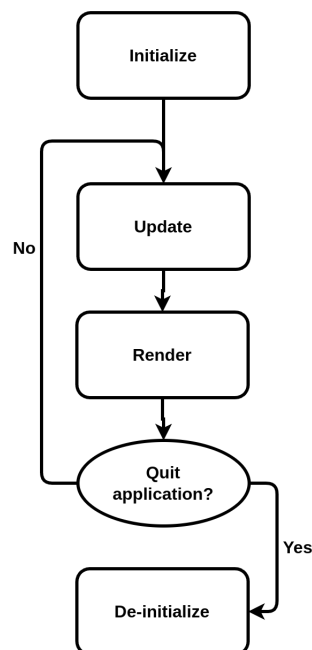
Capitolo 3

Design e implementazione

3.1 API

Di seguito è riportato il flusso di esecuzione di XrBridge:

Figura 3.1: Flusso di esecuzione di XrBridge



Lo schema riporta le operazioni fondamentali necessarie per sviluppare un'applicazione grafica.

Uno degli aspetti più importanti della API è la semplicità. È importante che l'utente che farà uso di XrBridge si possa concentrare il più possibile sullo sviluppare la propria applicazione invece di dover pensare ai dettagli di implementazione della libreria. Questo significa che la API deve esporre solamente metodi e parametri che sono assolutamente necessari per

l'utente e null'altro. Per questo motivo, è stato scelto di avere un metodo per ogni operazione fondamentale (TODO: vedi schema): inizializzazione, aggiornamento dello stato, render e de-inizializzazione. In realtà, aggiornamento e render potrebbero essere raggruppati in un'unica operazione, ma è stato deciso di lasciarli separati (se l'applicazione è in pausa e perciò non deve mostrare nulla, ma deve comunque rimanere in ascolto di eventi?). I dettagli della API e dei metodi si trovano nella documentazione apposita.

XrBridge è stato implementato sotto forma di una singola classe, dove ogni metodo pubblico rappresenta una delle operazioni fondamentali. Per facilitare l'utilizzo di XrBridge, oltre che ad offrire una documentazione più dettagliata della API e un'applicazione di esempio che ne dimostra chiaramente l'utilizzo, ogni metodo ha dei controlli per assicurarsi che essi vengano chiamati nell'ordine giusto (per esempio per assicurarsi che l'utente abbia inizializzato XrBridge prima di renderizzare una scena).

3.1.1 Inizializzazione

Il primo passo è quello di inizializzare XrBridge. Questo significa creare una connessione con la runtime di OpenXR, iniziare una sessione e caricare le estensioni necessarie. Questo passo è assolutamente necessario per poter utilizzare le funzionalità di OpenXR. Un errore durante questa fase è probabilmente causato da un problema con l'ambiente di esecuzione; potrebbe trattarsi di un errore di configurazione della runtime o un problema con l'hardware VR. Nessun'altra azione può essere eseguita prima dell'inizializzazione e l'inizializzazione può avvenire una sola volta (inizializzare più volte è considerato un errore).

XrBridge mette a disposizione il metodo `bool init(std::string& application_name)` per fare questo. Il parametro `application_name` non è di molta importanza e viene utilizzato da SteamVR solamente come dettaglio visivo e non ha alcun effetto sull'esecuzione dell'applicazione. Per i dettagli sul suo utilizzo, fare riferimento alla documentazione sulla API e all'applicazione di esempio.

3.1.2 Update

Questa è una delle operazioni che vengono eseguite di continuo per la durata dell'applicazione. Il metodo `bool update()` si occupa principalmente di gestire gli eventi generati da OpenXR, come ad esempio il termine di una sessione.

È inoltre importante per l'utente costantemente monitorare il valore di ritorno di questa funzione, poichè un errore potrebbe essere sollevato nel caso la runtime si disconnette (probabilmente l'utente ha disconnesso il suo visore o ha chiuso la runtime prima di terminare l'applicazione).

OpenXR fa distinzione fra un'istanza e una sessione. L'istanza è il collegamento alla runtime e viene creata una sola volta durante l'inizializzazione; mentre una sessione può essere terminata e ricreata più volte durante l'esecuzione dell'applicazione. Un'istanza viene ge-

neralmente terminata quando la runtime stessa viene terminata; in questo caso, XrBridge segnala un errore e necessita di un riavvio dell'applicazione. Nel caso che la sessione venisse terminata, invece, XrBridge attenderà semplicemente che la sessione venga ristabilita senza ritornare alcun errore o output.

3.1.3 Render

Questo è il passo più complesso di tutti e quello che differisce di più rispetto a OvVR. Come update, anche render viene seguito di continuo. Il rendering consiste nel generare le immagini che verranno poi proiettate sul visore VR seguendo le istruzioni dell'utente. L'utente invoca il metodo `bool render(render_function_t render_function)` per renderizzare. L'argomento `render_function` verrà spiegato più tardi.

OpenXR ha un sistema di rendering molto complesso che prende in considerazione la tempistica dei frame (refresh-rate), layers e swapchains.

OpenXR si aspetta che alcune azioni vengano eseguite entro un tempo limite stabilito dalla runtime. Nel caso il tempo limite scadesse, le conseguenze dipendono anche loro dalla runtime. Questo si applica anche al rendering dei frame. Nel caso di SteamVR, un mancato tempo limite potrebbe risultare in difetti grafici. Questa sincronizzazione è fatta dalla funzione `xrWaitFrame`.

Per costruire un frame, OpenXR utilizza un sistema di layers. Esistono diversi tipi di layers (3D, 2D, ...) che verranno poi sovrapposti (composizione) dalla runtime per generare l'immagine finale. È per esempio possibile avere un layer 2D dedicato alla GUI e un altro layer 3D per la scena. È anche possibile utilizzare un singolo layer 3D e disegnare la GUI manualmente senza affidarsi alla composizione offerta da OpenXR. Questo è l'approccio di XrBridge, dove viene utilizzato un singolo layer di proiezione (layer 3D).

Il layer di proiezione si aspetta un certo numero di *view*. Una *view* consiste in un'immagine che verrà mostrata sul dispositivo VR. Nel caso di un headset, ci saranno 2 *view*: una per l'occhio destro e una per l'occhio sinistro. È dunque necessario renderizzare la scena una volta per ogni occhio.

Una delle maggiori differenze tra OpenVR e OpenXR, quando si tratta di rendering, è chi si occupa di generare i framebuffer che verranno utilizzati per il rendering. Nel caso di OpenVR, è compito dell'utente creare i framebuffer, riempirli e inviarli a OpenVR una volta che il rendering è completo. Al contrario, OpenXR esige che sia la runtime a creare i framebuffer e che l'utente deve richiedere i framebuffer prima di poterli utilizzare. Questa differenza cambia in modo significativo la struttura dell'applicazione dal punto di vista del corso di realtà virtuale. Se prima gli studenti dovevano imparare il funzionamento per poter creare i propri framebuffer, ora questo processo è automatico.

Di seguito è riportata la sequenza di azioni richieste da OpenXR per renderizzare una scena. Le parti in *italico* sono implementate dall'utente, mentre il resto è eseguito automaticamente da XrBridge.

1. Attendere il momento giusto per iniziare a renderizzare il frame (sincronizzazione).
2. Preparare il rendering dell'intero frame (begin).
3. Preparare il rendering dell'occhio **sinistro** (begin).
4. *Renderizzare la scena dalla prospettiva dell'occhio sinistro.*
5. Finalizzare il rendering dell'occhio sinistro (end).
6. Preparare il rendering dell'occhio **destro** (begin).
7. *Renderizzare la scena dalla prospettiva dell'occhio destro.*
8. Finalizzare il rendering dell'occhio destro (end).
9. Finalizzare il rendering dell'intero frame (end).

Il frame è mostrato all'utente solamente alla fine di tutta la procedura di render, quando tutte le view sono complete.

Come si può notare, ci sono due sequenze di *begin* - ... - *end*: una per l'intero frame e una per ogni view/occhio.

Una possibilità di implementazione consiste nell'avere un metodo per ogni fase (*begin_frame*, *begin_eye*, *end_eye*, *end_frame*). Questa soluzione ha alcuni problemi: 1) una grande quantità di metodi sarebbero richiesti; 2) questo richiederebbe che l'utente faccia molta attenzione che ogni metodo sia chiamato nell'ordine corretto; 3) questo approccio non è scalabile (se si volesse sviluppare un'applicazione con una sola view?); 4) non è un approccio elegante; 5) è complesso da implementare.

Per XrBridge è stato scelto un approccio che appare molto più pulito, ordinato, semplice e scalabile: l'utente fornisce una propria funzione di render che verrà invocata quante volte necessario da XrBridge. È possibile utilizzare una lambda per questo, come nell'esempio seguente (semplificato):

```
void render()
{
    // ...

    xrbridge.render([&] (
        Eye eye,
        shared_ptr<Fbo> fbo,
        mat4 proj_matrix,
        mat4 view_matrix,
        uint32_t width,
        uint32_t height)) {
```



```
        // Render scene

    });

    // ...
}
```

Per più dettagli sul metodo e gli argomenti, fare riferimento alla documentazione di XrBridge. Lo svantaggio principale di questo approccio è il fatto che richiede conoscenza delle lambda, anche se minima.

Un grande vantaggio che si può notare dall'esempio è il fatto che, per ogni volta che la lambda viene chiamata, è facile reperire tutte le informazioni necessarie per renderizzare la scena: questo include il framebuffer da utilizzare, le matrici di proiezione e view e altri parametri che si possono aggiungere se desiderato senza necessitare di grandi modifiche al codice. Questo permette di semplificare la API ancora di più, dal momento che non sono necessari metodi appositi per recuperare queste informazioni. Ad ogni chiamata della lambda, tutte le informazioni necessarie sono subito a disposizione. Utilizzando la lista di cattura della lambda, catturando tutto come mostrato nell'esempio sopra, permette di eseguire operazioni di preparazione una sola volta (ad esempio costruire la lista di render) invece che sprecare risorse e ripetere le stesse operazioni per ogni chiamata della lambda. Se utilizzato in questo modo, dal punto di vista dell'utente, questo approccio diventa molto semplice da capire e implementare e permette di ignorare quasi completamente le complessità che nascono dall'utilizzo di lambda.

3.1.4 De-inizializzazione

Questo si tratta dell'ultimo passo, dopo il quale l'istanza di XrBridge diventerà inutilizzabile. A questo punto l'utente dovrà riavviare l'applicazione oppure creare una nuova istanza. Il metodo `bool free()` si occupa di terminare la sessione, l'istanza e liberare tutte le risorse utilizzate.

3.1.5 Gestione errori

Esistono svariati approcci alla gestione degli errori nel codice. I due approcci più puliti e ordinati sono *exceptions* e *errors as values*. Per questo progetto è stato scelto il secondo approccio, per alcuni motivi: 1) è lo stesso approccio utilizzato da OvVR; 2) è lo stesso approccio utilizzato da OpenXR; 3) è più semplice rispetto alle eccezioni; 4) nessuna interruzione inaspettata del flusso di esecuzione.

Tutti i metodi di di XrBridge seguono quindi la stessa convenzione: ritornano un valore `true` se il metodo è stato eseguito con successo e `false` in caso di errore. Inoltre, in caso di un errore, un messaggio che descrive l'errore viene stampato su `stderr`.

3.2 Strumenti e linguaggi di programmazione

Dal momento che XrBridge andrà a sostituire una libreria già in uso scritta in C++ e che i corsi di grafica e realtà virtuale fanno uso unicamente di C++, anche XrBridge verrà scritto in C++; nessun altro linguaggio di programmazione è necessario per la libreria.

Inoltre, nessuno strumento specifico è necessario per sviluppare la libreria. Ogni strumento aggiuntivo serve unicamente per sviluppare l'applicazione

3.3 OpenXR

3.3.1 Documentazione

OpenXR offre una documentazione molto estesa che descrive nei dettagli come la API deve essere utilizzata e come una runtime deve essere implementata. Ci sono due principali tipi di documentazione: un manuale mirato principalmente a chi desidera utilizzare OpenXR che spiega come fare uso della API OpenXR, e una specifica mirata a chi desidera implementare una runtime di OpenXR.

Il manuale della API è accessibile tramite il seguente link: <https://registry.khronos.org/OpenXR/specs/<VERSION>/man/html/>, dove VERSIONE è la versione di OpenXR che si sta utilizzando (Per esempio 1.0). Si può accedere velocemente al manuale di una funzione o struct specifica con un link del seguente formato: https://registry.khronos.org/OpenXR/specs/<VERSION>/man/html/<FUNZIONE_O_STRUCT>.html. Il manuale mostra informazioni come le definizioni delle funzioni e i loro parametri, possibili errori e una descrizione dettagliata della funzionalità.

La specifica di OpenXR è invece accessibile tramite il seguente link: <https://registry.khronos.org/OpenXR/specs/<VERSION>/spec/html/>. Come già detto, questo è utile principalmente per chi desidera implementare una runtime e non chi a chi semplicemente desidera utilizzare la API per sviluppare un'applicazione VR. Per questo progetto, ho usato molto raramente il documento di specifica.

OpenXR offre inoltre un tutorial ufficiale che spiega come sviluppare una semplice applicazione VR facendo uso di OpenXR. Il tutorial è accessibile al seguente link: <https://openxr-tutorial.com/> e permette di scegliere la combinazione di piattaforma e API grafica che si desidera utilizzare. Per questo progetto ho utilizzato Windows / OpenGL, dal momento che l'implementazione per Linux non differiva da quella di Windows. Ho seguito attentamente questo tutorial per comprendere il funzionamento di OpenXR e per sviluppare una semplice applicazione dalla quale ho poi estratto il codice necessario per sviluppare XrBridge.

3.3.2 Versioni

Al momento dello svolgimento di questo progetto, ci sono due versioni principali di OpenXR: 1.0 e 1.1. Le differenze principali tra queste due versioni sembrano minime e non importanti per questo progetto; ho scelto perciò di utilizzare la versione 1.0 per avere la massima compatibilità con le runtime.

Le principali differenze tra le due versioni sono due: sono stati apportati miglioramenti alla specifica di OpenXR e alcune estensioni sono state incluse in OpenXR core.

3.3.3 Runtime

OpenXR non si tratta di un software specifico, bensì di un'interfaccia standard di API che permette di sviluppare applicazioni di realtà aumentata per svariati dispositivi. Una runtime è semplicemente un software che implementa lo standard e offre alle applicazioni un'interfaccia con un dispositivo di realtà virtuale. È compito degli sviluppatori di dispositivi per realtà virtuale sviluppare una runtime per il proprio dispositivo. I requisiti di questo progetto richiedono che XrBridge debba funzionare almeno con SteamVR (una runtime di OpenVR e OpenXR sviluppata da Valve); dal momento che OpenXR è uno standard, XrBridge dovrebbe funzionare con tutte le altre runtime che implementano lo standard OpenXR correttamente, salvo per piccoli aggiustamenti.

Per permettere alle applicazioni di trovare la runtime corretta installata sul computer dell'utente, ogni runtime fa uso di un file manifest, ovvero un file di formato JSON che contiene alcune informazioni basi come il nome della runtime e il suo percorso nel filesystem. Questi file manifest sono generalmente installati in percorsi standard predefiniti (definiti dallo standard OpenXR) che dipendono dalla piattaforma; in alternativa è possibile specificare un percorso non-standard attraverso una variabile d'ambiente.

3.3.4 Loader

OpenXR utilizza un loader, ovvero un programma che ha lo scopo di individuare una runtime di OpenXR sul computer dell'utente. Questo programma viene sotto forma di una libreria che viene inclusa in un'applicazione VR. All'avvio dell'applicazione il loader cerca un file manifest di una runtime valido (TODO: chapter Runtime), stabilisce una connessione con la runtime e carica tutte le funzioni relative a OpenXR (run-time linking). Questo funziona in modo simile a librerie come GLEW per OpenGL.

OpenXR non offre un loader già compilato ed è quindi necessario compilarlo manualmente. L'applicazione Test include la libreria del loader già compilata e pronta all'uso. La versione di questa libreria è la 1.1.37. Di seguito sono riportate le istruzioni di come compilare la libreria loader per Windows utilizzando Visual Studio 2019 (per i seguenti passi è necessario Python 3; su Linux è possibile installare il loader utilizzando il package manager della distribuzione): 1. Scaricare e decomprimere il codice sorgente dal seguente

link: <https://github.com/KhronosGroup/OpenXR-SDK/releases/tag/release-1.1.37> (è possibile scegliere un'altra versione; per questo progetto solo la versione 1.1.37 è stata verificata). 2. Creare la cartella 'build/' nella root della repository. 3. Dall'interno di 'build/', creare il file di progetto di Visual Studio utilizzando cmake: `$ cmake -G 'Visual Studio 16 2019' -DDYNAMIC_LOADER=ON ..`. È possibile scegliere un'altra versione di Visual Studio. 4. Aprire la soluzione 'OPENXR.sln' appena creata con Visual Studio. 5. Scegliere la configurazione desiderata; per questo progetto è stato utilizzato *Release* e *x64*. 6. Compilare il progetto `openxr_loader`. 7. Recuperare i file generati dal percorso `build/src/loader/Release`. I file necessari sono `openxr_loader.dll` e `openxr_loader.lib`.

3.3.5 Filosofia della API

OpenXR segue una filosofia molto simile a Vulkan, essendo stato sviluppato dallo stesso gruppo. Ogni funzione di OpenXR è caratterizzata dal prefisso 'xr' seguito dal nome della funzione in camel case. Il passaggio di parametri alle funzioni di OpenXR viene fatto attraverso uno struct, il quale contiene un campo per ogni argomento che la funzione accetta e ha il prefisso `Xr`. Ogni funzione accetta almeno uno struct, anche se non necessita di parametri. Questo causa alcuni scenari interessanti, dove una funzione accetta come parametro uno struct senza membri. Ogni struct ha almeno due membri: `type`, il quale deve essere assegnato al momento della creazione dello struct e il suo valore è definito dallo standard OpenXR e `next`, il quale si tratta di un puntatore ad un altro struct in modo da permettere di estendere uno struct creando una catena. Nella maggior parte dei casi il membro `next` avrà un valore di `nullptr`.

```
// Inizializza tutti i campi dello struct al loro valore di default.
XrInstanceCreateInfo instance_create_info = {};
// Configura il tipo della struttura.
instance_create_info.type = XR_TYPE_INSTANCE_CREATE_INFO;
// Configura gli altri parametri ...
instance_create_info.createFlags = NULL;
// ...
```

3.3.6 Layers e estensioni

Similmente a Vulkan, OpenXR permette di estendere le proprie funzionalità attraverso dei layers e delle estensioni.

Un layer, come suggerisce il nome, è uno strato che può essere inserito fra OpenXR e l'applicazione. Un layer può offrire funzionalità come ad esempio tracing delle chiamate delle funzioni. Per questo progetto, non è stato necessario nessun layer. In caso si voglia aggiungere un layer, è sufficiente aggiungere il suo nome alla riga seguente all'interno di `xrbridge.cpp`, nella funzione `init`:

```
const std::vector<std::string> requested_api_layers = { };
```

Le estensioni, invece, permettono di aggiungere funzionalità extra a OpenXR. Alcune estensioni sono sviluppate da Khronos, ma possono anche essere sviluppate da terze parti. Per questo progetto, una singola estensione è stata necessaria: `XR_KHR_opengl_enable`. Come suggerisce il nome, questa estensione abilita il supporto per la API grafica OpenGL. In caso si voglia aggiungere un'altra estensione, è sufficiente aggiungere il suo nome alla riga seguente all'interno di `xrbridge.cpp`, nella funzione `init`:

```
const std::vector<std::string> requested_extensions = {
    "XR_KHR_opengl_enable",
};
```

3.3.7 Binding grafico

OpenXR fa uso di "graphic bindings", ovvero strutture che legano assieme una API grafica e una piattaforma. All'interno del codice, questi binding sono implementati sotto forma di struct. Esiste uno struct per ogni combinazione di API grafica (OpenGL, Vulkan, DirectX, ...) e piattaforma (Win32, X11, Wayland, ...) supportate e ogni struct richiede dei parametri legati alla piattaforma e alla API grafica scelta. Questi struct sono definiti nel file `openxr_platform.h` della libreria OpenXR. Ecco alcuni esempi:

API grafica	Piattaforma	Nome struct
OpenGL	Windows	<code>XrGraphicsBindingOpenGLWin32KHR</code>
OpenGL	Linux (Xlib)	<code>XrGraphicsBindingOpenGLXlibKHR</code>
OpenGL	Linux (Wayland)	<code>XrGraphicsBindingOpenGLWaylandKHR</code>

Durante l'inizializzazione di OpenXR, è compito dello sviluppatore istanziare e popolare uno di questi struct per la piattaforma che si vuole utilizzare. I requisiti richiedono solamente il supporto per OpenGL su Windows e Linux. Di seguito sono descritti gli approcci all'implementazione dei binding richiesti.

3.3.7.1 Binding OpenGL + Windows

Il binding OpenGL + Windows è rappresentato dalla struttura (`XrGraphicsBindingOpenGLWin32KHR`). Di seguito è riportata la definizione dello struct (i parametri `type` e `next` non sono importanti per questo capitolo):

```
typedef struct XrGraphicsBindingOpenGLWin32KHR {
    XrStructureType      type;
    const void* XR_MAY_ALIAS next;
    HDC                  hDC;
```

```

HGLRC                                hGLRC;
} XrGraphicsBindingOpenGLWin32KHR;

```

I due parametri importanti sono `hDC` e `hGLRC` e fanno parte della API Win32. `hDC` si tratta del *device context*, mentre *hGLRC* è il *OpenGL Rendering Context*; non è importante cosa rappresentato questi parametri, ma è necessario averli.

Qui si incontra un ostacolo: il contesto di OpenGL viene generato automaticamente da FreeGLUT. Per questioni di portabilità, FreeGLUT non offre un modo di recuperare questi parametri; a noi servono gli stessi parametri che FreeGLUT ha usato per creare il contesto OpenGL, il che significa che non possiamo generare un nuovo contesto/generare nuovi parametri. Fortunatamente, la API Win32 di Windows offre un modo per recuperare entrambi i parametri grazie alle funzioni `wglGetCurrentDC` e `wglGetCurrentContext` - queste due funzioni non accettano parametri. Grazie a queste funzioni, è possibile popolare tutta la struct e, di conseguenza, configurare OpenXR per questa piattaforma.

3.3.7.2 Binding OpenGL + Linux

Questo binding è più complesso rispetto a quello di OpenGL + Windows per due motivi.

La prima complicazione è dovuta al fatto che Linux non ha una API standard per quanto riguarda gli ambienti grafici. Al momento esistono infatti due principali piattaforme grafiche su Linux: X11 e Wayland. Inoltre, per X11, esistono approcci diversi per ognuna delle due librerie X11 più diffuse: Xcb e libX. Per questo motivo, esistono 3 struct per il binding OpenGL + Linux: `XrGraphicsBindingOpenGLXlibKHR` (LibX), `XrGraphicsBindingOpenGLXcbKHR` (Xcb) e `XrGraphicsBindingOpenGLWaylandKHR` (Wayland).

Inizialmente avevo deciso di utilizzare Wayland per due motivi. Il primo motivo è che lo struct per questo binding richiede un solo parametro, probabilmente rendendo più semplice l'implementazione. Il secondo motivo è il fatto che Wayland sta andando sempre di più a sostituire X11, tanto che molte delle distribuzioni di Linux principali supportano Wayland out-of-the-box. Lo svantaggio è il fatto che non è possibile eseguire applicazioni Wayland su X11. Questo approccio non è stato possibile a causa dell'assenza di supporto per Wayland in FreeGLUT. Nonostante FreeGLUT abbia un'implementazione di Wayland, come anche suggerito da un commento di uno degli sviluppatori (TODO: link to the comment), tale implementazione non è funzionante e sembra al momento abbandonata. A causa dei problemi appena menzionati, ho deciso di utilizzare l'approccio con X11.

Fortunatamente, esiste un software dal nome di XWayland che permette di eseguire quasi perfettamente molte applicazioni sviluppate per X11 in un ambiente Wayland. Usando l'approccio X11, quindi, potrò supportare entrambe le piattaforme X11 e Wayland.

Per quanto riguarda l'approccio X11, esistono due alternative: utilizzare la libreria Xlib (la libreria originale per interagire con X11) e Xcb (libreria alternativa a Xlib). Tra i due approcci,

utilizzare Xlib è il più semplice e diretto, poichè non richiede di stabilire una connessione a X11. Di seguito è riportato lo struct per il binding OpenGL + Linux (X11):

```
typedef struct XrGraphicsBindingOpenGLXlibKHR {
    XrStructureType          type;
    const void* XR_MAY_ALIAS next;
    Display*                 xDisplay;
    uint32_t                 visualid;
    GLXFBConfig              glxFBConfig;
    GLXDrawable              glxDrawable;
    GLXContext               glxContext;
} XrGraphicsBindingOpenGLXlibKHR;
```

I parametri `xDisplay`, `glxDrawable` e `glxContext` sono facilmente reperibili utilizzando le funzioni `glXGetCurrentDisplay`, `glXGetCurrentDrawable` e `glXGetCurrentContext` rispettivamente. Il problema sono i parametri `visualid` e `glxFBConfig`. Anche questi sono parametri che sono configurati da FreeGLUT ma non sono esposti e non esiste un modo per recuperarli come è possibile per i parametri precedenti.

3.3.8 Swapchain

Una swapchain in OpenXR è una serie di una o più immagini utilizzate per il rendering, dove il numero di immagini in una swapchain dipende dalla configurazione della runtime. Per esempio, se la runtime è configurata per utilizzare la tecnica *double buffering*, una swapchain sarà composta di 2 immagini, mentre se la runtime utilizza il *triple buffering*, le swapchain avranno 3 immagini.

Il formato delle immagini è definito dall'utente e dipende dalla API grafica utilizzata, mentre le immagini vere e proprie vengono create dalla runtime (questo è il comportamento opposto rispetto a OpenVR, dove è l'utente che si deve occupare di creare le immagini, renderizzarle e inviarle alla runtime). Durante la creazione di una swapchain l'utente può scegliere altri parametri, come ad esempio la dimensione delle immagini. Lo standard di OpenXR non specifica quali formati di immagini le runtime devono supportare e, perciò, l'utente deve scegliere un formato che è supportato dalla runtime che desidera utilizzare. In una swapchain, le immagini sono salvate in ordine e ogni immagine ha un indice che la distingue dalle altre immagini della stessa swapchain (effettivamente come un array).

Un'applicazione VR può avere una o più "view", dove una view consiste in una "schermata". Nel caso di un'applicazione che fa uso di un visore del tipo "head-mounted", esisteranno 2 view: una per ogni occhio. Nel caso di un sistema cave, esisterà una view per ogni lato. Ogni view ha una swapchain e, di conseguenza, il numero totale di immagini sarà $N_{immagini} = N_{view} \times k$, dove k è il numero di immagini per view (esempio: 2 nel caso di double-buffering). Per questo progetto, è richiesto che solamente 2 view siano supportate.

Quando è il momento di renderizzare, l'utente deve richiedere alla runtime quale immagine di una swapchain deve utilizzare; la runtime risponderà con un indice. Se, per esempio, una swapchain ha 3 immagini, la runtime ritornerà all'utente un indice 0, 1 o 2. L'utente può quindi utilizzare quella immagine per renderizzare. Una volta finito, l'utente "rilascia" l'immagine alla runtime che si occuperà poi di mostrare l'immagine sul visore VR. Al prossimo ciclo di rendering, la runtime fornirà un'altra immagine all'utente da utilizzare. Dalle mie osservazioni di SteamVR, l'ordine delle immagini fornite è: 0, 1, 2, 0, 1, 2, 0, 1, 2, ...

Per comodità, all'interno del codice, è stata definita una struttura creativamente chiamata `Swapchain` la quale aiuta a semplificare il codice legando assieme i vari componenti di una swapchain: la lista delle immagini, la dimensione delle immagini e la struttura `XrSwapchain` di OpenXR.

Le swapchain vengono istanziate quando una sessione di OpenXR inizia, all'interno del metodo `bool begin_session()`; una swapchain per view. Qui vengono create e configurate tutte le swapchain necessarie. Il metodo `shared_ptr<Fbo> create_fbo(GLuint w, GLuint h)` si occupa di generare le immagini (FBO = Frame Buffer Object) necessarie per le swapchain. Quest'ultimo fa affidamento alla classe `Fbo` già creata e utilizzata nel corso di realtà virtuale dal docente responsabile. Questa classe consiste in un wrapper attorno agli FBO di OpenGL e ne semplifica l'utilizzo. Grazie a questo, gli utenti avranno meno difficoltà quando dovranno utilizzare i framebuffer durante il rendering.

I framebuffer generati sono composti da una texture generata da OpenXR e il cui formato è specificato dall'utente (TODO: vedi capitolo Differenze fra piattaforme Windows e Linux) e un buffer di profondità.

3.3.9 Spazio di riferimento

In OpenXR, lo spazio di riferimento (`XrSpace`) è il sistema con cui un'applicazione VR tiene traccia della posizione e rotazione del mondo reale.

3.4 Limitazioni di FreeGLUT

FreeGLUT è una libreria multi-piattaforma che permette di gestire finestre, contesti OpenGL, mouse e tastiera. L'utilizzo di questa libreria è obbligatoria dal momento che viene utilizzata dal corso di grafica e il corso di realtà virtuale. Per questo progetto, è stata utilizzata la versione 3.6.0.

Come menzionato prima nel capitolo [Binding OpenGL + Linux], FreeGLUT non espone certi parametri che potrebbero essere necessari ad un'applicazione. È stato dunque necessario apportare modifiche alla libreria. Di seguito è descritto l'approccio scelto.

È stato aggiunto un nuovo file a FreeGLUT chiamato `freeglut_globals.h` e, al suo interno, sono state definite due variabili globali:


```
// Estratto da freeglut_globals.h
uint32_t freeglut_visualid = 0;
int freeglut_attributes[100] = { 0 };
```

All'interno di XrBridge, poi, includere questo file header e accedere alle variabili globali. Come parte della consegna del progetto, è anche presente una patch di git che descrive i cambiamenti effettuati nel dettaglio, così che possono essere facilmente applicati e analizzati.

Attenzione! La versione modificata di FreeGLUT è stata verificata solamente su Linux; per utilizzare FreeGLUT su Windows, utilizzare la versione originale non modificata.

Iniziando con il parametro più semplice: `visualid`. Questo si tratta di un semplice `uint32_t`. È bastato aggiungere una singola riga di codice che assegna il valore della variabile `visualInfo->visualid` alla variabile globale creata. Questo viene fatto all'interno della funzione `fgPlatformOpenWindow` nel file `src/x11/fg_window_x11.c`.

```
// src/x11/fg_window_x11.c @ 432
freeglut_visualid = visualInfo->visualid;
```

Il secondo parametro è `glxFBConfig`. Questo è più complicato rispetto al precedente, poiché non si tratta di un semplice valore che può essere facilmente copiato, bensì di una struttura interna che non viene esposta all'utente se non attraverso un puntatore. Questa struttura viene generata dalla funzione `glXChooseFBConfig`, la quale accetta una serie di attributi assieme ad altri parametri e ritorna un puntatore ad una struttura di tipo `GLXFBConfig`. La soluzione è fortunatamente piuttosto semplice: è sufficiente ottenere la lista di attributi usati per generare la struttura e invocare nuovamente il metodo `GLXFBConfig`. Per fare questo, è stato necessario salvare gli attributi in una delle variabili globali appena create. Questo è stato fatto nella funzione `fghChooseConfig` nel file `src/x11/fg_window_x11_glx.c`. Infine, all'interno di XrBridge, è bastato chiamare la funzione `glXChooseFBConfig` con gli attributi corretti e così si ottiene la struttura necessaria.

```
// src/x11/fg_window_x11_glx.c @ 102
for (unsigned int i = 0; i < 100; ++i)
{
    freeglut_attributes[i] = attributes[i];
}
```

Una libreria alternativa a FreeGLUT è GLFW, la quale espone i parametri interni attraverso dei metodi specifici. Come menzionato all'inizio del capitolo, però, non è stato possibile utilizzare questa libreria.

3.5 Differenze fra piattaforme Windows e Linux

Ci sono alcune importanti differenze fra le piattaforme Windows e Linux che hanno richiesto aggiustamenti all'interno del codice. In precedenza è già stato menzionato come il binding grafico differisce tra piattaforme (TODO: link to chapter); di seguito saranno presentate alcune differenze che hanno causato difficoltà nello sviluppo di XrBridge.

Quando si tratta di generare i framebuffer per ogni view/occhio, è possibile specificare il formato dell'immagine desiderato (es: GL_RGBA16F, GL_SRGB8, ...). Il formato dipende fortemente dalla API grafica scelta e la runtime, dal momento che OpenXR non specifica alcun formato di default.

Una delle difficoltà riscontrate durante lo sviluppo, è il fatto che SteamVR supporta formati diversi a seconda della piattaforma. Di seguito sono elencati i formati supportati da SteamVR sulle piattaforme principali dalla versione attuale di SteamVR:

Formato	Windows	Linux
GL_RGBA16F	Sì	No
GL_SRGB8	Sì	Sì

SteamVR è costantemente aggiornato e, perciò, è molto probabile che in futuro i formati supportati cambino.

A causa di queste differenze, è stato deciso di permettere all'utente di specificare il formato per piattaforma invece di avere una singola configurazione globale.

Nella sezione di configurazione di XrBridge (all'inizio del file `xrbridge.cpp`) è possibile specificare il formato con le variabili `XRBRIDGE_CONFIG_SWAPCHAIN_FORMAT_WINDOWS` e `XRBRIDGE_CONFIG_SWAPCHAIN_FORMAT_LINUX`.

Capitolo 4

Risultati

Capitolo 5

Conclusioni

5.0.1 Applicazione di test

Dal momento che il progetto consiste nello sviluppare una libreria, sarà necessario un'applicazione che farà uso di XrBridge. Questa applicazione avrà il doppio scopo di aiutare nello sviluppo e test della libreria e

5.1 Sezione

Donec et nisl id sapien blandit mattis. Aenean dictum odio sit amet risus. Morbi purus. Nulla a est sit amet purus venenatis iaculis. Vivamus viverra purus vel magna. Donec in justo sed odio malesuada dapibus. Nunc ultrices aliquam nunc. Vivamus facilisis pellentesque velit. Nulla nunc velit, vulputate dapibus, vulputate id, mattis ac, justo. Nam mattis elit dapibus purus. Quisque enim risus, congue non, elementum ut, mattis quis, sem. Quisque elit. Esempio di citazione [1], [2, 3], [4].^{1 2}

¹Questa è una nota a piè di pagina.

²Questa è un'altra nota a piè di pagina.

Bibliografia

- [1] C. Yuen and A. Oudalov. The feasibility and profitability of ancillary services provision from multi-microgrids. In *Power Tech, 2007 IEEE Lausanne*, pages 598 –603, july 2007.
- [2] H. Farhangi. The path of the smart grid. *Power and Energy Magazine, IEEE*, 8(1):18 –28, january-february 2010.
- [3] L.H. Tsoukalas and R. Gao. From smart grids to an energy internet: Assumptions, architectures and requirements. In *Electric Utility Deregulation and Restructuring and Power Technologies, 2008. DRPT 2008. Third International Conference on*, pages 94 –98, april 2008.
- [4] F. Blaabjerg, R. Teodorescu, M. Liserre, and A.V. Timbus. Overview of control and grid synchronization for distributed power generation systems. *Industrial Electronics, IEEE Transactions on*, 53(5):1398 –1409, oct. 2006.