#### ESC 23T05: DirectX user-land (DXGI e D3D10)

1. Introduzione

Buon pomeriggio, eccomi quest’anno con un altro talk riguardante cose mediamente strane.

Oggi voglio portarvi il risultato di qualche settimana di studio e ricerca e di come ho, in parte, compreso il funzionamento di alcuni componenti interni di Windows.

~~Prima di tutto vorrei iniziare questa presentazione con una citazione di Windows che mi ha accompagnato durante lo sviluppo di queste cose.~~

**\*cambio slide\***

~~Questa immagine la ho vista un po’ di volte avviando windows per testare questi componenti.~~

Come scritto nella slide sono Christian Rendina, sono uno studente di 5° superiore un po’ cresciuto. Sono un programmatore e reverse engineer improvvisato appassionato di basso livello, computer graphics e sviluppo di videogiochi.

Quello che voglio portarvi parte da una curiosità, come funziona internamente DirectX 10? Lo stack di Windows Vista è davvero cosi diverso da Windows XP? DirectX 10 è davvero un cambiamento rispetto al 9?

**\*slide nuova\***

1. Background DX

Partiamo prima dalle basi, che cos’è DirectX? Si tratta un set di componenti per sviluppare videogiochi, l’immagine che vedere nella slide ne fa vedere alcuni (di cui in realtà sono quasi tutti deprecati), i componenti che ci focalizzeremo oggi si tratta di Direct3D. La sua controparte su Linux può essere o OpenGL, la versione 2 è relativa a Direct3D9, mentre le versioni moderne a Direct3D10 o 11, e Vulkan il quale è relativo di Direct3D12.

1. Background stack video

L’architettura di DirectX è direttamente collegata al modo in cui funziona lo stack video di Windows e uno dei cambiamenti più grandi introdotti per il funzionamento di DirectX moderno si tratta della parte relativa nel kernel.

Per poter capire qual è stato il grande cambiamento introdotto su Windows Vista, tocca prima percorre come funzionava il sistema video di Windows prima dell’introduzione di Windows Vista.

**\*slide nuova\***

Quello che vedete nella slide è il sistema grafico di Windows XP denominato XP Display Driver model o XDDM. Quello che caratterizza l’XDDM si tratta nel dividere la gestione della scheda grafica in due parti principali programmabili dal venditore della scheda grafica. La prima è il driver Miniport, questo driver è ciò che effettivamente aggiorna il monitor dello schermo e svolge alcune funzioni come di enumerazione dei vari monitor collegati. Il secondo driver principale è chiamato Display driver, questo driver è quello che consente di effettuare l’accelerazione grafica effettiva come renderizzare un modello 3d. Entrambi questi driver girano nel kernel e parlano tra di loro con un metodo di comunicazione chiamato IOCTL, praticamente un metodo (diciamo cosi) per gestire input e output di un dispositivo.

Tutti e due questi driver sono collegati nel kernel attraverso il componente di Windows chiamato win32k.sys. Questo componente include la gestione completa dei processi di Windows tra cui tutta la parte di disegno, denominata GDI, che è responsabile di disegnare la finestra, i bordi della finestra, i vari elementi o il desktop. Questo componente su Linux lo ritroviamo su X11 o Wayland dove risiede nella parte utente, quella dove girano le nostre applicazioni.

**\*slide nuova\***

Quello che vedete in slide, si tratta della architettura grafica di Windows moderna chiamata Windows Display Driver Model o WDDM. Questa architettura è ciò che si basa sulle DirectX moderne (dalla 10 in poi).

Il driver del miniport e il display driver sono ancora rimasti in questo modello, però, mentre il driver del miniport continua ad essere nel kernel, il display driver è stato spostato fuori dal kernel, nella parte dell’utente, quella dove poi girerà la nostra applicazione. Tutte le parti di funzionamento e di collegamento di questi due driver, precedentemente gestiti attraverso il sottosistema di Windows win32k, sono stati sostituiti con un driver apposito chiamato dxgkrnl.sys, il quale gestisce praticamente tutto il funzionamento del sistema Video; difatti, il display driver e il miniport driver comunicano attraverso di esso e non più in maniera diretta.

Il sistema di disegno di Windows, denominato GDI e win32k sono ancora presenti nel sistema e svolgono ancora la loro stessa funzione, ma non gestiscono più le schede grafiche. Il compito di parlare direttamente con la scheda grafica spetterà SOLO al Miniport driver e a nessun altro. Ovviamente, win32k e, di conseguenza GDI, dovranno parlare la parte DirectX nel kernel per fornire accelerazione, usando la GPU, del disegno dei bordi delle finestre e cosi via.

Questo cambiamento di separare la gestione delle schede grafiche dal sistema di Windows ad un modulo a parte è stato sicuramente un enorme cambiamento, che avvicina Windows al funzionamento di Linux.

**\*slide successiva\***

Quello che vedete nella slide, questo è il funzionamento dello stack video di Linux.

Nella realtà, ci sono molte similitudini a WDDM e quello usato su Linux. Per prima cosa, abbiamo anche qui l’esistenza di un driver video che gira all’interno dello spazio dell’utente, caratterizzato da Mesa.

Mesa al suo interno implementa le varie runtime o interfacce per parlare con la scheda grafica, ovvero OpenGL o Vulkan, che come citato prima sono i rispettivi di DirectX e la parte video per poter parlare nelle GPU come DirectX, la grande differenza tra DirectX e Linux sta nel fatto che c’è una totale divisione tra la parte user creata dai creatori di GPU e dalle runtime di base di DirectX.

All’interno del kernel, anche Linux ha una specie di sottosistema DirectX chiamato Direct Rendering Manager o DRM, esso comunica direttamente con la scheda grafica attraverso un driver creato dai produttori di GPU. Possiamo comprare il funzionamento di DRM come una specie di dxgkrnl e driver miniport integrati in un unico sistema. All’interno di DRM è contenuto anche KMS che è un semplice modulo per poter prendere informazioni sulla scheda video, su Windows questa parte è integrata su dxgkrnl ma esposta, come vedremo in un altro modulo della parte utente.

Ovviamente su Linux è possibile usare altri componenti per poter usare le varie runtime come OpenGL creati dai produttori stessi di schede grafiche, come nel caso di nVidia mentre su Windows tutti sono obbligati a seguire il funzionamento di DirectX.

Per quanto riguarda la parte utente, dove ho svolto la maggior parte dei miei studi, sono composti da due grandi componenti. La runtime di Direct3D, come d3d10 o d3d11 che parla con il display driver per poter inviare comandi alla GPU per, es. renderizzare un cubo e DirectX Graphics Infrastructure o DXGI, che, come anticipato prima, espone su Windows varie funzionalità, che vedremo in dettaglio, per l’enumerazione di monitor o schede video. Questo è, infatti, KMS ma su Windows

1. Componenti userland

**\*slide successiva\***

L’ultima cosa che dobbiamo sapere, ma un piccolo accenno è come Windows permette di far comunicare più componenti contemporaneamente.

Questa tecnologia è chiamata Component Object Model, o COM, detta in parole povere consente di implementare una serie di interfacce che vengono rese disponibili in ogni programma attraverso componenti interni di Windows. Queste interfacce sono definibili e implementabili completamente dai programmatori e sono inclusi all’interno di componenti come alcune DLL esempio comdlg32.

**\*slide successiva\***

Finalmente finito tutta questa noiosa parte introduttiva, possiamo finalmente parlare di come funzionano realmente DXGI e le runtime di DirectX.

Questo diagramma che vedere nella slide creato da me, descrive davvero come le DirectX comunica all’interno del sistema operativo.

Diviso totalmente, DXGI, l’api di enumerazione comunica con le runtime DirectX con delle strutture interne e non documentate, i quali utilizzano GDI per poter saltare dalla zona utente a quella del kernel. La zona del kernel passa attraverso WIN32K, il sottosistema di Windows per poi comunicare con il sistema DirectX nel kernel.

**\*slide successiva\***

Microsoft, quindi, non è stata totalmente onesta nel loro diagramma di funzionamento di WDDM, siccome tutto è collegato assieme, come una forte colla liquida, alla antica parte di WIN32K o GDI. Infatti, il componente utente di GDI, espone tutte le funzioni del sistema DirectX da poter utilizzare dalle varie applicazioni. Ogni API del sistema di DirectX, prefissati con D3DKMT, sono implementati all’interno di WIN32K, il quale poi reindirizzerà ogni cosa al componente kernel di DirectX DXGKRNL, dove avverranno realmente tutte le varie operazioni.

1. Inizio vero del talk: spiegazione di DXGI

In queste settimane mi sono focalizzato sul componente base delle runtime DirectX, ovvero DXGI. Cosa è effettivamente implementato all’interno di DXGI?

**\*slide successiva\***

DXGI espone una classe madre, chiamata Factory, che offre come ponte di accesso per poter creare e usare ogni altro componente. DXGI implementa la rappresentazione di una scheda grafica, chiamata Adapter, e ci consente di poter implementare e prendere alcune informazioni da essa. Ogni scheda grafica, ovviamente, è collegata con un monitor, il monitor è implementato in una classe chiamata Output e ci consente, oltre di prendere risoluzione e cosi via, anche altre informazioni utili come la parte dove poter disegnare.

DXGI inoltre, implementa una swapchain che, come spiegerò meglio dopo, serve come componente base per poter renderizzare su schermo, alcune classi base per capire e gestire i componenti e una serie di strutture interne non documentate che consentono lo scambio di informazioni con altri componenti come il compositor grafico DWM.

**\*slide successiva\***

Infatti, ogni computer può avere una o più schede grafiche montate nel proprio PC rappresentate da un adapter, il quale può avere uno o più monitor collegati ad esso, dove al suo interno possono esserci una o più applicazioni che possiamo accedere per poter renderizzare magari il nostro gioco. Ovviamente DXGI supporta anche la GPU remote, come quelle utilizzate nei Server.

Come viene implementato tutto questo brodo?

Teoricamente, da come Microsoft ha descritto WDDM, dovrebbero esserci chiamate specifiche all’interno del sistema di DirectX che permettono, ad esempio, di poter vedere tutti i Monitor collegati una scheda grafica, e magari, la parte di collegamento con GDI e WIN32K è stata fatta per legacy, esempio per supportare i vecchi giochi.

**\*slide successiva\***

Quello che vedete qui, si tratta del codice di enumerazioni di ogni scheda grafica collegata nel computer. Notate qualcosa di strano?

**\*slide successiva\***

Si, internamente DXGI in realtà non fa altro che mappare le risorse di GDI, quelle utilizzate in precedenza con Windows XP e chiamare una funzione del kernel per poter iniziare ad accedere alla scheda grafica partendo dalla risorsa legacy. E questa mentalità si estende anche nell’implementazione dei monitor o dei monitor remoti, Microsoft potrebbe, in realtà, averci mentito, siccome le runtime di DirectX in realtà dipendono molto dal funzionamento delle risorse di XDDM sfasando il reale mito di divisione completa tra il sistema nuovo e vecchio, infatti, è per questo motivo che alcune funzionalità di GDI sono rimaste invariate e DXGI offre semplicemente un involucro più moderno.

**\*slide successiva\***

Il componente, in realtà più importante per renderizzare, il quale ho accennato precedentemente è la Swapchain. Componente che, nella teoria, è molto semplice.

La swap chain non è altro che un componente che tiene al suo interno varie copie dello schermo, la sua responsabilità principale è di scambiare queste copie tra di loro dopo che le operazioni di rendering sono completate. Questo serve per evitare tearing e problemi simili, in soldoni noi non vorremmo mai vedere metà di un cubo roteante renderizzato nella finestra e l’altra metà no. Vorremmo che ciò che è presente nello schermo venga aggiornato solo quando la rotazione del cubo viene effettivamente completata, non mischiata con una parte della rotazione precedente. Con il metodo della swapchain questo è possibile. La swap chain tiene in memoria lo schermo principale visualizzato a schermo, chiamato Frontbuffer, e uno o più copie di esso dove siamo pronti per disegnare, chiamati BackBuffer.

Per quanto sia una cosa semplice, su Windows mi ha dato parecchi problemi per poter capirla a fondo.

**\*slide successiva\***

Windows infatti, non ha un solo tipo di swapchain, ma ben 4 e probabilmente altre che ancora non ho compreso. Il primo tipo, denominato da me DWM in onore del compositor grafico, serve per aggiornare lo schermo in caso di una applicazione che giri a schermo intero, come l’appunto compositor grafico.

Il secondo tipo di swapchain è denominato DDA ma, non ho molte informazioni a riguardo siccome fa parte di una cosa non documentata.

La terza swapchain denominata da me HWND ovvero Handle to Windows, che nelle api di Windows un HWND rappresenta una finestra. Questa swapchain è utilizzata per aggiornare una finestra associata, come ad esempio un gioco in finestra.

La quarta swapchain denominata Partner, fa parte del componente di DirectComposition, che consente di poter accelerare trasformazioni di immagini con effetti o animazioni.

L’ultima swapchain da me conosciuta chiamata XAML è quella che consente di aggiornare su schermo le applicazioni di Windows 8 dello store oppure le applicazioni UWP ovvero quelle di Windows 10.

Ognuna di queste poi si va a scomporre in alcune cose particolari, come aggiornare lo schermo senza il compositor oppure gli schermi remoti esempio via RDP. C’è molto di più di quanto sembri su Windows per poter disegnare qualcosa su schermo, infatti non ho ancora totalmente compreso tutto il funzionamento di questo componente.

**\*slide successiva\***

1. L’UMD? Come DXGI si comunica con esso

Il secondo componente del lato utente di DirectX sono le runtime effettive, quelle che il nostro gioco usa per mandare comandi come disegnare un cubo ovvero Direct3D 10, 11 o 12.

Al loro interno sono implementati il resto delle funzionalità che consentono di gestire in memoria le risorse come texture o vertex buffer, il collegamento con il display driver, compilazione di shader da inviare nella GPU, generazione di modelli nello schermo e cosi via.

Si tratta di un componente molto grande che, nonostante ci siano versioni differenti, hanno in comune molti pezzi che sono stati poi estesi tra le nuove versioni. Io ho specificamente studiato DirectX 10, ma molti concetti e funzionalità sono applicabili anche nel 11 o 12.

Ma come fa DirectX10 a parlare con il driver userland?

**\*slide successiva\***

La comunicazione avviene attraverso una serie di callback, ovvero funzioni che vengono passati tra i due moduli. La runtime di DirectX espone usa serie di questi callback che vengono poi inviati all’UMD per seguire ogni serie di funzione. Ovviamente anche le runtime sono in grado di parlare con l’UMD.

Questi non sono gli unici callback che l’UMD ha accesso, per avere accesso diretto alle funzionalità del kernel, DXGI carica e poi passa con l’uso di strutture interne una serie di callback esposti dal kernel per poter mandare direttamente comandi a DirectX chiamati DXGI Thunks. Questa lista di callback varia in base alle varie versioni di Windows utilizzate e si può utilizzare per sapere quali funzionalità il driver UMD e le runtime di DirectX possono e non possono utilizzare dal kernel.

Nel diagramma si parlare anche dell’utilizzo di DXGI, ma DXGI non dovrebbe gestire l’UMD giusto?

**\*slide successiva\***

Questo diagramma spiega la relazione tra l’UMD, DXGI e le runtime di DirectX.

Come prima cosa, il nostro gioco chiede alle runtime di DirectX 10 di creare un nuovo dispositivo per poter renderizzare. Le runtime chiederanno a DXGI quale è la scheda grafica predefinita per poter disegnare su schermo, e DXGI manderà sia la GPU predefinita che le relative informazioni del Monitor. Le runtime, successivamente, chiederanno a DXGI di caricare l’UMD associato, questo serve perché ogni GPU può avere un UMD diverso. DXGI caricherà solo il componente ma non darà effettivamente l’ok per poter iniziare ad usare le funzionalità dell’UMD.

L’informazione dell’UMD viene passata poi dalla runtime, che darà l’ok per iniziare nell’effettivo a poter usare questa accelerazione grafica assieme alla lista di informazioni necessarie all’UMD per avviarsi. Una volta che il driver è pronto, comunicherà l’OK alle runtime che, a loro volta, comunicheranno l’OK al gioco dandogli effettivamente la possibilità di iniziare a renderizzare.

**\*slide successiva\***

1. Tutto molto figo ma non c’è già Wine?

Molti di questi concetti sono implementati anche all’interno di Wine. Ma date le differenze sostanziali tra Wine e Windows, Wine non ha alcuna parte del kernel implementato, infatti DXGI e DirectX sono implementate con un layer di conversione chiamato WineD3D, che reindirizza le chiamate a OpenGL o Vulkan nelle nuove versioni.

Wine però, seguendo il funzionamento di Windows, implementa la comunicazione tra le sue runtime DirectX e DXGI con lo stesso metodo utilizzando da Windows. Questo potrebbe, almeno in teoria, significare che potremmo rimpiazzare un componente o l’altro in maniera compatibile a Wine se implementiamo le sue interfacce interne.

**\*slide successiva\***

1. Live dimostrazione: spiegaizone del mio PoC

Come dimostrazione pratica, voglio farvi vedere in azione il risultato di questi studi. Quella che vedete qui è una macchina virtuale con Windows 7 Service Pack 1 installato su VMWare. VMWare implementa l’accelerazione grafica per DirectX11. Ho sostituito il componente di DXGI con una mia versione creata da zero frutto die miei studi. Come si può vedere, questo componente riscritto è già in grado, seppur in modo primitivo, di far vedere qualcosa su schermo e poterci giocare un po’, il tutto 100% compatibile con Windows.

**\*slide successiva\***

1. Outro

Ci sono molti componenti che ancora non si conoscono totalmente e solo DXGI include più di 20 api non documentate e mai studiate, sicuramente c’è molto altro riguardante il funzionamento di DirectX12 e cosi via, penso di aver solo scavato la superfice di un sistema molto più grande.

**\*slide successiva\***

Ringrazio a tutti i partecipanti dell’ESC per avermi seguito verso questo viaggio alla scoperta di DirectX su Windows. Ringrazio ovviamente l’End Summer Camp per avermi invitato ad esporre un talk anche quest’anno e le persone del discord di ReactOS Longhorn per avermi aiutato nel decifrare molte di queste cose interne. La live demo che avete visto, assieme all’implementazione di DXGI è disponibile sotto licenza open source alla repository di github presente in slide.