Alma Mater Studiorum • Università di Bologna

SCUOLA DI SCIENZE

Corso di Laurea Triennale in Informatica

vitaQuakeIII: Porting di un engine di gioco opensource da PC su console portatile

Relatore:
Prof. Ivan Lanese

Presentata da:
Alessio Tosto

Anno Accademico 2018 - 2019

Sessione I

Abstract

Lo studio qui illustrato si basa su di un progetto applicativo denominato vitaQuakeIII, un'applicazione che permette di eseguire codice scritto tramite engine di gioco id-Tech 3, disponibile per PC, su un dispositivo sul quale tale engine non è ufficialmente supportato (Sony PlayStation® Vita).

Verranno illustrate le maggiori problematiche dovute dall'operazione di porting e come queste verranno risolte. La limitatezza delle risorse della console in termini di memoria disponibile e velocità di calcolo rispetto ad un PC è uno dei problemi fondamentali affrontati durante il documento. Inoltre verranno trattate le operazioni necessarie ad adattare il codice di id Tech 3 per quanto riguarda l'input, il codice di rete ed il renderer (parte dell'engine relativa alla stampa a schermo di frame).

Indice

1 Introduzione

1.1 Contesto

Sony PlayStation® Vita [8] è una console di gioco handheld entrata in commercio il 17 dicembre 2011. Da diversi anni è possibile eseguire su di essa codice non firmato scritto da sviluppatori amatoriali e diversi appassionati hanno iniziato a sviluppare applicazioni per essa.

id Tech 3 è un motore di gioco (software per permettere la creazione in maniera semplificata di videogiochi occupandosi della fisica di gioco, logica di gioco, rendering, etc. permettendo allo sviluppatore di concentrarsi sulle meccaniche specifiche del gioco come la creazione di mappe, modelli 3D da utilizzare, textures, etc.) per PC sviluppato da id Software nel 1999. Inizialmente a codice sorgente chiuso, è stato ufficialmente rilasciato in maniera opensource sotto licenza GPLv2 il 19 agosto 2005. Tra le varie innovazioni introdotte rispetto agli engine precedenti della stessa famiglia, vi sono miglioramenti nella qualità delle animazioni mediante l'utilizzo di vertici e tecniche di interpolazione, un miglioramento nella struttura dei modelli 3D supportati rendendo possibile il rendering di superfici curve realistiche, l'aggiunta di nebbia volumetrica ed un sistema di shading (tecnica per il quale vengono eseguiti diversi step di rendering per applicare effetti sulle superfici di un modello in modo che rispondano in maniera differente in base alle luci che li colpiscono) sviluppato su di OpenGL 1.x.

vitaQuakeIII è nato allo scopo di rendere possibile eseguire applicazioni sviluppate tramite id Tech 3 su di una Sony PlayStation® Vita abilitata all'esecuzione di codice non firmato. Per far si che tutto questo sia possibile, è indispensabile un lavoro di adattamento del codice in quanto id Tech 3 utilizza API differenti da quelle esposte da VitaSDK [13] (SDK amatoriale per lo sviluppo di applicazioni per Sony PlayStation® Vita; verrà approfondito in un paragrafo successivo), inoltre la macchina ospitante è sprovvista di un mouse ed una tastiera per l'input e non supporta nativamente OpenGL per l'utilizzo della GPU. Il lavoro di porting sarà svolto prendendo come base ioquake3 [3], un source port di Quake III: Arena per Windows, Linux e macOS progettato per essere di facile utilizzo per la creazione di nuovi videogiochi grazie a nuove funzioni specifiche che semplificano alcuni passaggi della creazione di giochi tramite engine id Tech 3 ma, soprattutto, esso è stato progettato per essere relativamente semplice da portare su di nuove piattaforme.

1.2 Teoria alla base di vitaQuakeIII

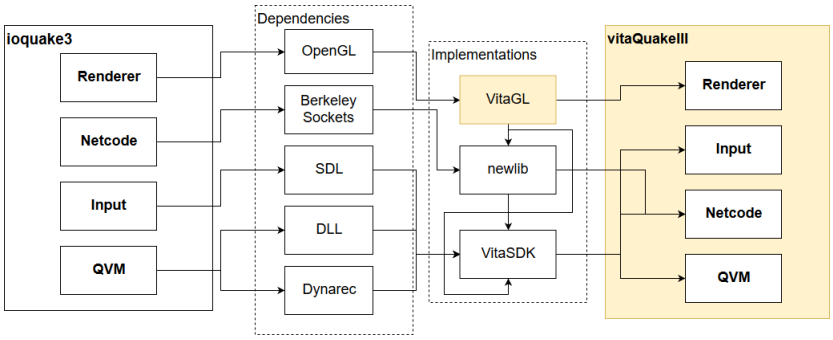

Figura 1: Struttura del lavoro di porting effettuato

La fig. 1 mostra il lavoro di porting eseguito che verrà trattato in questa tesi e come le varie componenti principali di **ioquake3** verranno **implementate** tramite **VitaSDK** in maniera diretta o tramite l'utilizzo di librerie ausiliarie (**newlib** e **VitaGL**) a seconda delle **dipendenze** richieste. Preso il codice sorgente di ioquake3, quest' ultimo, mediante una ricompilazione, sarà fruibile su una macchina ospitante originariamente non supportata dall'engine. Sorge però un problema quando la macchina ospitante è sprovvista di alcune librerie essenziali per la compilazione del codice. VitaSDK, toolchain non ufficiale utilizzato per la creazione di applicazioni per Sony PlayStation® Vita in linguaggio C/C++, fornisce un port di newlib [6], un aggregato di diverse librerie rilasciate con licenza open source, permettendo di fruire di funzioni basilari disponibili normalmente su sistemi Linux implementando, quindi, ad esempio, i meccanismi di allocazione dinamica della memoria (malloc, memalign, free, etc...) e di gestione delle stringhe (strlen, strstr, strcmp, etc...). Ciononostante, rimangono ancora diversi problemi irrisolti per arrivare a poter considerare la compilazione:

1. Il netcode originariamente utilizzante socket standard GNU/Linux non disponibile in maniera completa sull'implementazione fornita da VitaSDK di newlib.
2. Il codice riguardante l'input che fa riferimento a mouse e tastiera, non disponibili su di una Sony PlayStation® Vita.
3. L'assenza di un meccanismo di caricamento di librerie dinamiche.
4. La mancanza di supporto per OpenGL da parte del VitaSDK.

Per ovviare a questi problemi, parte del codice sorgente di ioquake3 dovrà essere adattato per utilizzare le funzioni esposte da VitaSDK inoltre, bisogna realizzare un wrapper per l'utilizzo di OpenGL sviluppato sulle API grafiche esposte da VitaSDK per l'utilizzo della GPU (SceGxm).

Un'altra problematica di cui tener conto è la limitazione in risorse dovuta dalla piattaforma ospitante. Differentemente da un PC o smartphone, Sony PlayStation® Vita risulta essere piuttosto limitata nelle risorse disponibili per quanto riguarda potenza di calcolo e memoria disponibile per lo sviluppatore.

1.3 Obiettivo

Nel corso di questa tesi vedremo come è stato sviluppato vitaQuakeIII e come il lavoro svolto possa essere utilizzato per redigere delle linee guida usufruibili in vista di altri lavori di software porting. Verrà illustrato inoltre come è stato realizzato il wrapper con interfaccia OpenGL sviluppato su di SceGxm denominato VitaGL e come quest'ultimo sia utilizzabile per altri progetti affini a vitaQuakeIII.

Prima di procedere nei dettagli implementativi, verrà proposta una breve introduzione per comprendere nel dettaglio cosa sia una Sony PlayStation® Vita, cosa sia ioquake3 e quale sia la sua struttura, come lavorare con VitaSDK e quali siano le differenze tra OpenGL e SceGxm.

2 Background

Vogliamo ora introdurre in breve, ma in modo preciso, alcuni dei concetti, delle definizioni e degli strumenti rivelatisi fondamentali e importanti per la realizzazione e per lo sviluppo di vitaQuakeIII. Questa breve introduzione sarà necessaria, inoltre, per la piena comprensione del documento.

2.1 Sony PlayStation® Vita

Figura 2: Foto frontale di una Sony PlayStation® Vita

Sony PlayStation® Vita [8] è una console handheld di gioco sviluppata da Sony e messa in commercio il 17 dicembre 2011. Essa possiede una CPU ARM Cortex TM-A9 Quadcore underclockata ad un clock variabile tra 333MHz a 444MHz seppure tale clock può essere aumentato fino ad un massimo di 494 MHz tramite overclocking. La GPU utilizzata è una PowerVR SGX543MP4+ con 128 MB di VRAM e clock da 222 MHz. Dispone di 512 MB di RAM di cui 128 MB riservati al sistema operativo.

La console è provvista di 12 tasti fisici programmabili (freccette direzionali, i classici tasti Croce, Quadrato, Triangolo, Cerchio, i tasti Start e Select e i dorsali laterali sinistro e destro), 4 tasti fisici non programmabili (i due tasti di regolazione del volume, il tasto Power riservato al sistema operativo per l'accensione, la sospensione e lo spegnimento della console ed il tasto PlayStation riservato al sistema operativo per la sospensione dell'applicazione attualmente in esecuzione), sensori di movimento (accelerometro e giroscopio a 3 assi) ed un pannello touch sul retro della console di tipo capacitivo multitouch utile se si vuole utilizzare anche il retro della console per alcune funzioni specifiche ad esempio la gestione della camera in giochi in prima persona. Inoltre lo schermo (di risoluzione 960x544) è anch'esso touch di tipo capacitivo multitouch.

2.2 Rendering

Uno dei punti più complessi che verrà trattato in questa tesi è il lavoro svolto per l'adattamento del renderer di ioquake3 in modo da funzionare su Sony PlayStation® Vita.

Un renderer è la parte di un'applicazione che si occupa della stampa a schermo di frame. Esso può essere implementato tramite CPU, assumendo la denominazione di renderer software, o tramite GPU, ed assumendo così la denominazione di renderer hardware. Entrambe le implementazioni presentano vantaggi e svantaggi, infatti, un renderer software sarà semplice da estendere e da adattare per nuove piattaforme ma sarà, generalmente, più lento in quanto l'intero carico di lavoro dell'engine sarà posto sulla CPU mentre un renderer hardware sarà più complesso da modificare tuttavia sarà molto più performante in quanto il carico di lavoro dell'engine verrà diviso tra CPU e GPU.

Entrando più in dettaglio nei renderer hardware (in quanto id Tech 3 è sprovvisto di renderer software), nella maggior parte dei casi, viene utilizzata per la sua implementazione una API denominata OpenGL, che verrà approfondita in un paragrafo successivo. Le due fasi principali di un renderer hardware eseguite dalla GPU sono la rasterizzazione dei vertici ed il fragment processing.

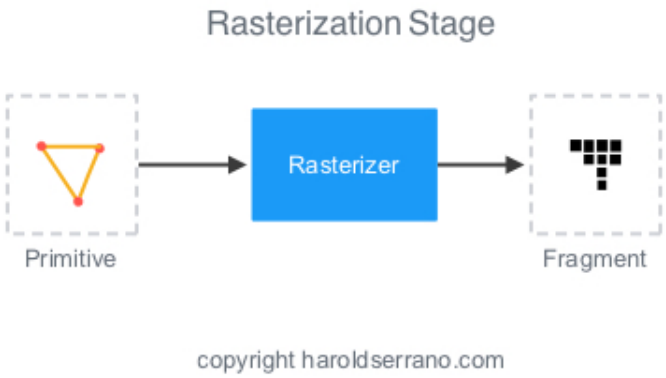

Figura 3: Rasterizzazione dei vertici in un renderer hardware

Per rasterizzazione dei vertici, come mostrato nella fig. 3, si intende il processo per il quale, dati dei vertici definiti in uno spazio tridimensionale ed una primitiva (triangoli, quadrati, linee, etc), essi verranno trasformati in una serie di frammenti che rappresenteranno la regione delimitata dalla figura definita dai dati forniti. Durante questa fase verranno inoltre calcolate le colorazioni di ogni frammento, infatti, lo sviluppatore può fornire un vettore di colori per definire tale proprietà per ogni vertice oppure può fornire un vettore di coordinate che, in combinazione con una texture, verranno utilizzate dal rasterizer per determinare il colore di ogni frammento. Al termine di questa fase, i frammenti prodotti come output saranno poi processati durante la seconda fase denominata, appunto, fragment processing nella quale, se richieste dallo sviluppatore, possono essere eseguite operazioni aggiuntive come, ad esempio, l'**alpha testing**, ossia un controllo del valore alpha del colore di un frammento in modo che se quest'ultimo risulti maggiore o minore di un dato valore specificato dallo sviluppatore, il frammento in

questione verrà scartato. Al termine dell'esecuzione di queste operazioni, i frammenti che non sono stati scartati, verranno infine stampati su schermo.

E' possibile eseguire operazioni aggiuntive che manipolano i frammenti da processare anche prima della fase di fragment processing ad esempio tramite l'utilizzo del **viewport**, ossia un'operazione che permette di definire un'area bidimensionale sottoinsieme dell'area definita dallo schermo e permette di utilizzare quest'ultima che target della stampa del frame, traslando, pertanto, tutti i frammenti processati, o tramite l'utilizzo di **scissor testing** e **depth range**, ossia due operazioni che permettono di definire rispettivamente un'area bidimensionale x, y, w, h ed un range massimo di profondità z1, z2 in modo che i frammenti risultato della rasterizzazione verranno immediatamente scartati se non fanno parte di questo spazio 3D che viene a formarsi tramite le due tecniche in questione o tramite l'uso del **culling**, processo per il quale frammenti che sono nascosti da altri modelli nello spazio 3D del frame vengono scartati prima del fragment processing in modo da ridurre il carico di lavoro della GPU.

Nei renderer hardware moderni, entrambe le fasi descritte sono implementate tramite shader (veri e propri programmi scritti in un linguaggio di programmazione ed eseguiti unicamente dalla GPU). Pertanto, il termine **shading** viene utilizzato per definire il processo mediante cui vengono applicate proprietà aggiuntive ad un modello 3D mediante l'esecuzione di operazioni aggiuntive durante la rasterizzazione (vertex shading) o durante il fragment processing (fragment shading) come, ad esempio, con il **texture environment** ossia un processo che permette di alterare in maniera dinamica la resa grafica delle superfici a seconda di alcuni effetti preimpostati.

2.3 ioquake3

ioquake3 è un motore di gioco opensource sviluppato partendo dal codice sorgente di Quake III: Arena rilasciato da id Software nel 2005. Sebbene supporti nativamente Quake III: Arena e Quake III: Team Arena, l'engine è stato modificato in modo da poter essere utilizzato con relativa semplicità per la creazione di progetti nuovi. Particolare enfasi è stata posta anche nella riscrittura del codice in modo da essere portato su altre piattaforme in maniera agevole. Il codice di ioquake3 è suddiviso in un eseguibile principale (che per comodità chiameremo quake3.exe) e tre virtual machine (denominate QVM) ognuna delle quali svolge un compito specifico durante l'esecuzione di quake3.exe.

2.3.1 Quake Virtual Machine (QVM)

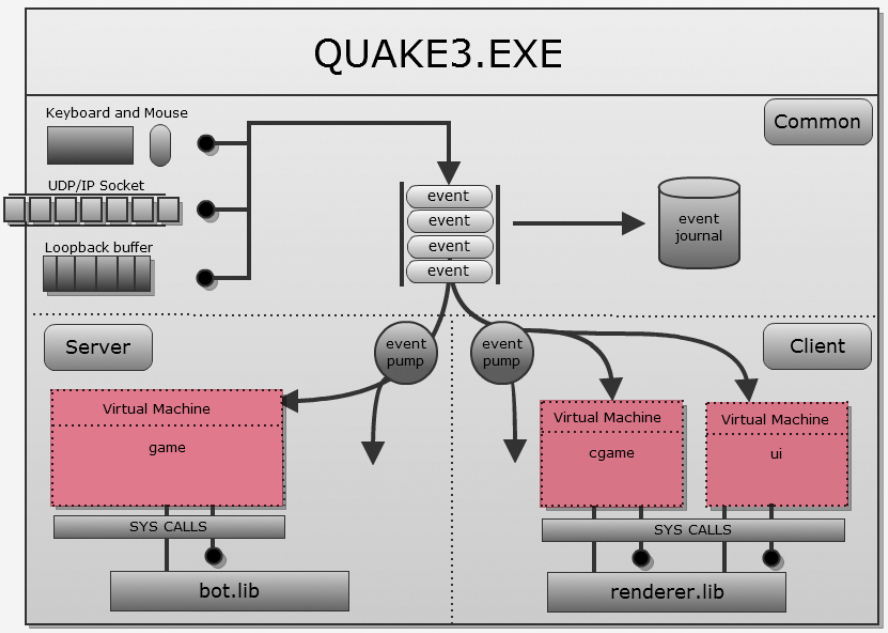

Figura 4: Composizione di quake3.exe

Da come si evince dall'analisi del codice sorgente di Quake 3 di Fabien Sanglard[9] schematizzata nella fig. 4, le tre vitual machine utilizzate in ioquake3 sono **cgame**, la quale riceve messaggi durante il gameplay e si occupa di operazioni di rendering 3D, **ui**, la quale riceve messaggi durante la fase di utilizzo dei menù di gioco e si occupa di operazioni di rendering 2D ed infine **game**, la quale riceve costantemente messaggi e si occupa della logica di gioco e dell'intelligenza artificiale dei bot.

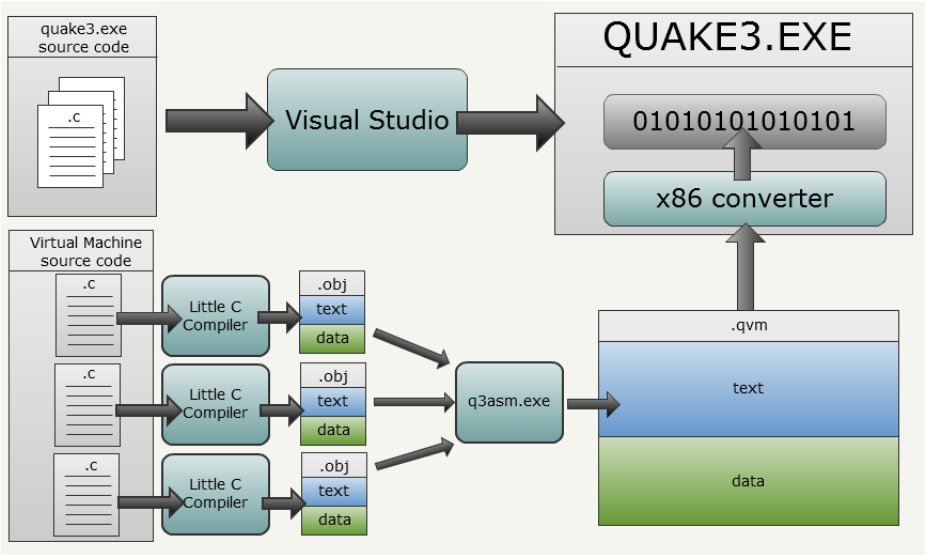

Figura 5: Struttura interna di una QVM

Come mostra la fig. 5, quake3.exe è provvisto di un interprete di bytecode (il quale si differenzia dal bytecode utilizzato in un interprete Java ed assume il nome di VM bytecode) utilizzato per l'esecuzione di syscall chiamate dalle virtual machine. Quest'ultime, sebbene composte da codice C, vengono compilate tramite LCC (Little C Compiler) al fine di non generare un eseguibile finale bensì solamente la loro rappresentazione interna (sezioni text, data e bss con simboli di import/export) come mostrato nella fig. 6 sottostante.

```
data
export variableC
align 4
LABELV variableC
byte 4 0
export fooFunction
code
proc fooFunction 4 4
ADDRFP4 0
INDIRP4
ARGP4
ADDRLP4 0
ADDRGP4 strlen
CALLI4
ASGNI4
ARGP4 variableA
INDIRI4
ADDRLP4 0
INDIRI4
ADDI4
RETI4
LABELV $1
endproc fooFunction 4 4
import strlen
bss
export variableB
align 4
LABELV variableB
skip 4
import variableA
```

Figura 6: Esempio di VM bytecode per una QVM

Tramite un tool realizzato da id Software denominato **q3asm**, tali sezioni vengono assemblate in un file .qvm che verrà, a runtime, caricato da quake3.exe in memoria ed eseguito tramite l'interprete prima citato. L'esecuzione di tali virtual machine può avvenire in pura interpretazione o tramite conversione del bytecode in istruzioni per la macchina ospitante. Quest'ultima modalità di esecuzione, denominata dynarec (Dynamic Recompilation), seppure produca performance migliori, risulta essere più soggetta ad errori e può causare inaspettati crash di quake3.exe durante l'esecuzione.

Per ovviare ai problemi di stabilità introdotti dall'esecuzione mediante dynarec e alla lentezza di compilazione introdotta dall'utilizzo di tool esterni creati da idSoftware (ad esempio **q3asm**), quest'ultima aggiunse inoltre una terza modalità di esecuzione delle virtual machine. Quest'ultima si rivelerà essere la scelta adottata in vitaQuakeIII poichè unisce la velocità della ricompilazione dinamica alla stabilità dell'interprete. Tale modalità riguarda la compilazione del codice delle virtual machine sottoforma di librerie dinamiche (DLL su Windows) così che il codice sia già pronto per l'esecuzione sulla macchina ospitante saltando completamente il passaggio per l'interprete di bytecode.

2.3.2 Renderer

Il renderer dell'id Tech 3 si occupa della fase di rendering (spiegata nel dettaglio nel paragrafo 2.2) del motore di gioco. Esso supporta modelli 3D in formato MD3 i quali permettono l'utilizzo di animazioni realistiche e la generazione di superfici curve realistiche mediante l'utilizzo di vertici. Differentemente dagli engine precedenti di casa id Software, presenta due sostanziali differenze: l'assenza di un renderer software e la presenza di un sistema di shading costruito su di OpenGL 1.x [4].

Quest'ultimo è un'innovazione particolarmente interessante poichè la fixed function pipeline (set di stati configurabili durante il rendering di un frame che costituiscono lo stato attuale della fase di rendering e la configurazione attuale della macchina a stati) proposta da OpenGL 1.x non supporta l'utilizzo di shader (feature introdotta con OpenGL 2.0). L'introduzione di questo sistema di shading permise ai programmatori dell'epoca di avere un tool molto potente per l'applicazione di effetti speciali ed il miglioramento dell'apparenza delle superfici dei modelli 3D utilizzati ma, ovviamente, il tradeoff è costituito dal fatto che, per ogni shader applicato ad una superficie, il mondo di gioco verrà riprocessato un'altra volta. In sostanza, l'utilizzo di shader in maniera eccessiva avrebbe potuto portare ad un degrado significativo delle performance su macchine dei primi anni 2000.

Inoltre, l'assenza di un renderer software implica la necessità per la macchina ospitante di possedere una GPU che supporti OpenGL 1.x in quanto il rendering tramite CPU risulta essere fuori discussione. Nel nostro caso, VitaSDK espone solamente SceGxm come libreria per l'utilizzo della GPU, pertanto, per ovviare al problema, si è ricorsi alla creazione di una libreria intermedia che implementasse la fixed function pipeline insieme alle funzioni e feature basilari di OpenGL 1.x utilizzate da id Tech 3 tramite SceGxm. Questa libreria intermedia verrà poi chiamata VitaGL.

2.4 VitaSDK

Per poter parlare di VitaGL, è necessario introdurre VitaSDK[5], il toolchain non ufficiale realizzato da sviluppatori amatoriali per creare eseguibili per Sony PlayStation® Vita. Esso è composto principalmente da quattro componenti principali:

1. Una versione adattata dei tool proposti dal GNU Embedded Toolchain for Arm (arm-none-eabi) in modo da essere pienamente utilizzabile per la creazione di eseguibili adatti ad una Sony PlayStation® Vita. Il prefisso di tali tool è stato cambiato in arm-vita-eabi per distinguersi dal toolchain ufficiale (arm-vita-eabi-gcc, arm-vita-eabi-g++, etc...).

2. Tool aggiuntivi per la creazione di file ausiliari necessari per la creazione di un eseguibile compatibile con Sony PlayStation® Vita. (ad es. **vita-makesfoex** utilizzato per la generazione del file SFO necessario per ogni applicazione il quale contiene alcuni metadata relativi all'applicativo come la versione del software, il suo nome, il suo livello di parental control, etc.)

3. Un rudimentale package manager, denominato **vdpm**, per l'installazione, in maniera semplificata, di librerie necessarie allo sviluppo.

4. Un database YAML [14] contenente tutti i Name Identifier (in breve NID, verranno approfonditi nel prossimo paragrafo) necessari alla creazione delle librerie stub utilizzate in fase di linking. Quest'ultimo è correlato con dei file header che espongono e documentano le funzioni utilizzabili tramite le suddette librerie stub.

2.4.1 Name Identifier (NID)

I NID sono numeri interi a 32 bit utilizzati per identificare moduli, librerie e funzioni generati tramite hashing SHA256 di alcuni dati unici a seconda di quale tipo di NID si sta prendendo in esame (unique data per i moduli, nome della libreria per le librerie e nome della funzione per quest'ultime). Come specificato in precedenza, VitaSDK dispone di un database YAML contenente i NID necessari all'utilizzo delle funzioni esposte dal sistema operativo della console.

```yaml
SceGxm:
  nid: 0x0D0AA0CB
  libraries:
    SceGxm:
      kernel: false
      nid: 0xF76B66BD
      functions:
        sceGxmAddRazorGpuCaptureBuffer: 0xE9E81073
        sceGxmBeginCommandList: 0x944D3F83
        sceGxmBeginScene: 0x8734FF4E
        sceGxmBeginSceneEx: 0x4709CF5A
        sceGxmColorSurfaceGetClip: 0x07DFEE4B
        sceGxmColorSurfaceGetData: 0x2DB6026C
        sceGxmColorSurfaceGetDitherMode: 0x200A96E1
        sceGxmColorSurfaceGetFormat: 0xF3C1C6C6
        sceGxmColorSurfaceGetGammaMode: 0xEE0B4DF0
        sceGxmColorSurfaceGetScaleMode: 0x6E3FA74D
        sceGxmColorSurfaceGetStrideInPixels: 0xF33D9980
        sceGxmColorSurfaceGetType: 0x52FDE962
        sceGxmColorSurfaceInit: 0xED0F6E25
        sceGxmColorSurfaceInitDisabled: 0x613639FA
        sceGxmColorSurfaceIsEnabled: 0x0E0EBB57
        sceGxmColorSurfaceSetClip: 0x86456F7B
        sceGxmColorSurfaceSetData: 0x537CA400
        sceGxmColorSurfaceSetDitherMode: 0x45027BAB
```

Figura 7: Esempio di un modulo nel database YAML del VitaSDK

Il database, del quale è possibile vedere i NID di un modulo in fig. 7, è utilizzato da **vita-libs-gen**, uno dei tool proposti da VitaSDK, per generare librerie di stub utilizzate per il linking statico durante la creazione di eseguibili per la console. Tali NID verranno poi risolti nelle vere funzioni al lancio dell'applicazione o (in caso di import deboli, ossia funzioni importate a runtime mediante il caricamento di moduli ausiliari similarmente a come funziona con le DLL su sistemi Windows) al lancio dei moduli del quale le funzioni fanno parte direttamente dal sistema operativo.

2.4.2 SceGxm

SceGxm è la libreria ufficiale per l'utilizzo della GPU su Sony PlayStation® Vita. Essa dispone, similarmente a come succede con OpenGL, di una macchina a stati per la gestione di diverse funzionalità della GPU tuttavia, a differenza di OpenGL 1.x, non presenta una fixed pipeline bensì un sistema basato su shader come si potrebbe trovare in OpenGL 2, inoltre, differentemente da come accade in OpenGL 2, SceGxm non presenta un runtime shader compiler il che significa che gli shader non possono essere modificati durante l'esecuzione del programma bensì possono essere caricati solamente come binari precompilati. E' possibile tuttavia modificare il binding degli attributi esposti dagli shader ed i settaggi di blending (funzionalità per il quale, data una tipologia di blending, un colore ed una texture, i frammenti durante il fragment processing verranno stampati in maniera diversa poichè verrà applicato il colore desiderato sulla texture mediante l'algoritmo descritto dal tipo di blending scelto) da utilizzare con questi ultimi mediante un sistema di runtime patching implementato in SceGxm (sceGxmShaderPatcher).

Vi sono inoltre altre differenze sostanziali tra OpenGL e SceGxm che dovranno essere gestite e risolte da VitaGL: l'assenza di alpha testing e texture environment, la presenza di scissor testing solo tile based (è possibile effettuare scissor testing solamente per dimensioni multiple di 8 pixel in altezza e larghezza a differenza di OpenGL che permette scissor testing pixel based) ed infine l'utilizzo di algoritmi differenti per il viewport ed il depth range.

2.5 OpenGL

OpenGL [7] è la più famosa API ideata per l'utilizzo della GPU per il processo di rendering. E' considerata uno standard ed è adottata dalla maggioranza delle case del settore. Sono disponibili diverse versioni di OpenGL e l'ultima disponibile è la 4.6. Per i sistemi embedded, è disponibile una specifica più ristretta della libreria denominata OpenGL ES (Open Graphics Library for Embedded Systems). OpenGL è sprovvista di codice sorgente essendo quest'ultima solo una specifica adottata dai principali produttori di schede grafiche.

La versione al quale VitaGL punta maggiormente è la specifica OpenGL ES 1.1. Essa è sprovvista di rendering tramite shader ed implementa una fixed function pipeline correlata con una macchina a stati per la gestione delle feature da utilizzare durante la fase di rendering.

Alcune feature della specifica OpenGL ES 2.0 sono state implementate in VitaGL anche se quest'ultime non sono feature richieste per lo sviluppo di vitaQuakeIII e pertanto non verranno prese in esame durante la scrittura di questo documento.

3 VitaGL, l'implementazione

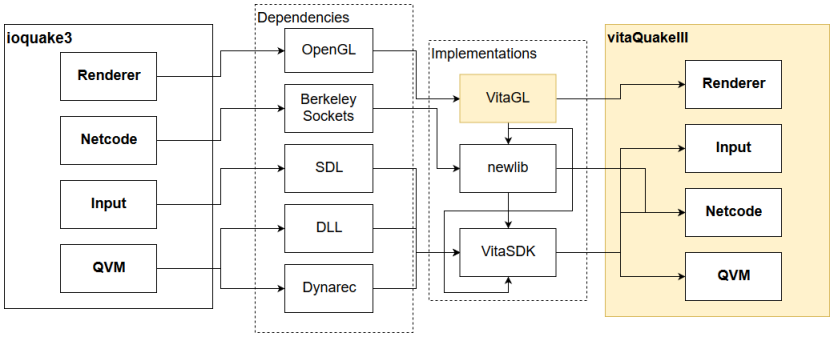

Figura 8: Struttura del lavoro di porting effettuato

VitaGL è la libreria intermedia sviluppata utilizzando SceGxm che permetterà l'utilizzo in maniera indiretta di OpenGL su Sony PlayStation® Vita e, come mostrato nella fig. 8, è il principale componente per quanto riguarda il porting del codice relativo al rendering di ioquake3. Il suo sviluppo è iniziato il 29 Dicembre 2017 ed è tutt'ora in corso sebbene sia già utilizzabile per progetti non estremamente complessi.

L'obiettivo finale della libreria è di fornire un'implementazione completa e performante di OpenGL 1.x ed OpenGL 2.x. L'enfasi attuale è riposta nell'implementazione di feature senza fare estrema attenzione alle performance (che vengono comunque prese in esame in fase di sviluppo).

Allo stato attuale essa implementa un buon numero di feature di OpenGL 1.x ed alcune feature core di OpenGL 2: depth testing, alpha testing, scissor testing pixel based, culling, rendering tramite fixed function pipeline o tramite shader personalizzati, funzioni basilari di texture environment, etc.

Una release ufficiale non è disponibile per via della natura fortemente sperimentale della libreria ma il codice sorgente è completamente fruibile su GitHub [11].

VitaGL è completamente scritta in C (7400 righe di codice circa), con l'eccezione di alcuni shader, del quale parleremo più avanti, scritti in Cg (243 righe di codice), un linguaggio sviluppato da NVIDIA con l'idea di essere simile al C ma utilizzabile per sviluppare programmi per la GPU [1]. Essa presenta alcune funzioni specifiche per l'inizializzazione di sceGxm ed alcune funzioni uniche, non presenti nell'originale specifica di OpenGL, per la gestione del rendering (operazioni che normalmente vengono gestite da SDL per applicazioni utilizzanti OpenGL, come ad esempio l'aggiornamento della schermata dell'applicazione).

```
1 // vgl*
2 void vglEnd(void);
3 void *vglGetTexDataPointer(GLenum target);
4 void vglInit(uint32_t gpu_pool_size);
5 void vglInitExtended(uint32_t gpu_pool_size, int width, int height, \
6     int ram_threshold, SceGxmMultisampleMode msaa);
```

```
void vglMapHeapMem(void);
size_t vglMemFree(vglMemType type);
void vglStartRendering(void);
void vglStopRendering(void);
void vglStopRenderingInit(void);
void vglStopRenderingTerm(void);
void vglUpdateCommonDialog(void);
void vglUseVram(GLboolean usage);
void vglWaitVblankStart(GLboolean enable);
```

Codice 1: Prototipi delle funzioni uniche di VitaGL.

Le funzioni non facenti parti della specifica di OpenGL, come mostrato nel cod. 1, hanno come suffisso **vgl** per differenziarsi dalle normali funzioni di OpenGL con suffisso **gl**.

Per l'implementazione delle funzioni standard di OpenGL, VitaGL presenta una macchina a stati che si occupa della gestione delle molteplici feature attivabili e personalizzabili tramite le funzioni proposte da OpenGL (glEnable, glDisable, glBlendFunc, etc...). Ogni volta che una di queste funzioni viene chiamata, lo stato della macchina viene alterato ed i settaggi di sceGxm vengono aggiornati di conseguenza.

```
static void update_polygon_offset(){
        switch (polygon_mode_front){
        case SCE_GXM_POLYGON_MODE_TRIANGLE_LINE:
                if (pol_offset_line)
                    sceGxmSetFrontDepthBias(gxm_context, \
                        (int)pol_factor, (int)pol_units);
                else sceGxmSetFrontDepthBias(gxm_context, 0, 0);
                break;
        case SCE_GXM_POLYGON_MODE_TRIANGLE_POINT:
                if (pol_offset_point)
                    sceGxmSetFrontDepthBias(gxm_context, \
                        (int)pol_factor, (int)pol_units);
                else sceGxmSetFrontDepthBias(gxm_context, 0, 0);
                break;
        case SCE_GXM_POLYGON_MODE_TRIANGLE_FILL:
                if (pol_offset_fill)
                    sceGxmSetFrontDepthBias(gxm_context, \
                        (int)pol_factor, (int)pol_units);
                else sceGxmSetFrontDepthBias(gxm_context, 0, 0);
                break;
        }
        switch (polygon_mode_back){
        case SCE_GXM_POLYGON_MODE_TRIANGLE_LINE:
                if (pol_offset_line)
                    sceGxmSetBackDepthBias(gxm_context, \
                        (int)pol_factor, (int)pol_units);
                else sceGxmSetBackDepthBias(gxm_context, 0, 0);
```

```
28                 break;
29         case SCE_GXM_POLYGON_MODE_TRIANGLE_POINT:
30                 if (pol_offset_point)
31                     sceGxmSetBackDepthBias(gxm_context, \
32                         (int)pol_factor, (int)pol_units);
33                 else sceGxmSetBackDepthBias(gxm_context, 0, 0);
34                 break;
35         case SCE_GXM_POLYGON_MODE_TRIANGLE_FILL:
36                 if (pol_offset_fill)
37                     sceGxmSetBackDepthBias(gxm_context, \
38                         (int)pol_factor, (int)pol_units);
39                 else sceGxmSetBackDepthBias(gxm_context, 0, 0);
40                 break;
41         }
42 }
43
44 void glPolygonOffset(GLfloat factor, GLfloat units){
45         pol_factor = factor;
46         pol_units = units;
47         update_polygon_offset();
48 }
```

Codice 2: Implementazione di glPolygonOffset in VitaGL.

Un esempio è dato dal cod. 2 che mostra l'implementazione di glPolygonOffset e come al variare di settaggi in OpenGL corrispondano variazioni nella macchina a stati di VitaGL (righe 45-46) e come queste variazioni vengono poi utilizzate per alterare i settaggi di sceGxm (righe 2-41).

Tramite questo meccanismo, la maggior parte delle funzioni può essere implementata tuttavia vi sono alcuni passaggi che meritano un approfondimento sulle scelte implementative adottate.

Vedremo ora come le numerose discrepanze funzionali tra OpenGL e SceGxm sono state risolte durante lo sviluppo di VitaGL rendendo possibile la nascita di vitaQuakeIII.

3.1 Fixed Function Pipeline

Come spiegato precedentemente, OpenGL 1.x, utilizzato da id Tech 3 per il renderer, non supporta shader personalizzati bensì presenta una fixed function pipeline che si occupa di gestire, mediante vari stati programmabili, tutti gli effetti da applicare durante il rendering. Per ovviare a questo problema, VitaGL presenta alcuni shader di default che verranno utilizzati nel caso in cui shader personalizzati non siano richiesti. Tali shader, tramite alcune uniform (variabili globali di uno shader al quale è possibile assegnare un valore tramite codice C dall'applicazione), emulano il comportamento della fixed function pipeline presente in OpenGL 1.x e pertanto permettono l'utilizzo

di alcune feature che normalmente non sono disponibili con sceGxm (allo stato attuale, VitaGL implementa, tramite questo meccanismo, i sistemi di texture environment, alpha testing e fogging).

```
float4 main(
        float2 vTexcoord : TEXCOORD0,
        float4 vColor : COLOR,
        float vFog : FOG,
        uniform sampler2D tex,
        uniform float alphaCut,
        uniform int alphaOp,
        uniform int texEnv,
        uniform int fog_mode,
        uniform float4 fogColor,
        uniform float4 texEnvColor
        )
{
        float4 texColor = tex2D(tex, vTexcoord);

        // Texture Environment
        if (texEnv < 4){
                if (texEnv == 0){ // GL_MODULATE
                        texColor = texColor * vColor;
                }else if (texEnv == 1){ // GL_DECAL
                        texColor.rgb = lerp(vColor.rgb, texColor.rgb, texColor.a);
                        texColor.a = vColor.a;
                }else if (texEnv == 2){ // GL_BLEND
                        texColor.rgb = lerp(vColor.rgb, texEnvColor.rgb, texColor.rgb);
                        texColor.a = texColor.a * vColor.a;
                }else{ // GL_ADD
                        texColor.rgb = texColor.rgb + vColor.rgb;
                        texColor.a = texColor.a * vColor.a;
                }
        }

        // Alpha Test
        if (alphaOp < 7){
                if (alphaOp == 0){
                        if (texColor.a < alphaCut){
                                discard;
                        }
                }else if (alphaOp == 1){
                        if (texColor.a <= alphaCut){
                                discard;
                        }
                }else if (alphaOp == 2){
                        if (texColor.a == alphaCut){
                                discard;
                        }
```

```
46                  }else if (alphaOp == 3){
47                          if (texColor.a != alphaCut){
48                                  discard;
49                          }
50                  }else if (alphaOp == 4){
51                          if (texColor.a > alphaCut){
52                                  discard;
53                          }
54                  }else if (alphaOp == 5){
55                          if (texColor.a >= alphaCut){
56                                  discard;
57                          }
58                  }else{
59                          discard;
60                  }
61          }
62
63          // Fogging
64          if (fog_mode < 3){
65                  texColor.rgb = lerp(fogColor.rgb, texColor.rgb, vFog);
66          }
67
68          return texColor;
69  }
```

Codice 3: Fragment Shader per texture con tinta di VitaGL

Il codice mostrato qui sopra (cod.3) mostra uno dei fragment shader (codice eseguito dalla GPU per ogni frammento da processare dopo la rasterizzazione dei vertici eseguita da un vertex shader) di default utilizzati da VitaGL. Da come si evince da esso, una volta venuti a conoscenza degli algoritmi implementativi utilizzati da OpenGL per le feature richieste [2], è possibile applicare tali conoscenze negli shader in modo da ricreare la fixed function pipeline utilizzata da OpenGL 1.x. Da notare come l'implementazione di nuove feature richieda di aggiungere branching negli shader stessi infatti lo shader mostrato in figura presenta 3 branch ossia quello relativo all'implementazione del texture environment (righe 16-30), quello relativo all'alpha testing (righe 32-61) e quello relativo al fogging (righe 63-66). Considerando che i fragment shader vengono eseguiti per ogni frammento da processare, aggiungere molte feature può risultare degradante per le performance della libreria in quanto, differentemente da come accade con del normale codice C, gli shader non vengono ottimizzati in fase di compilazione pertanto le if in cascata potranno arrivare ad essere eseguite per un numero estremamente alto di volte per frame in base a quanto complessa è la scena da renderizzare. Un modo per evitare ciò sarebbe quello di suddividere gli shader in più file a seconda di quali feature sono attive e quali non lo sono nella pipeline (Ad esempio, creare uno shader che non presenti il branching relativo al fogging se quest'ultimo non è attivo). Allo stato attuale però, tale possibilità non è stata presa in considerazione per via del fatto che le feature implementate sono ancora relativamente poche; inoltre l'assenza di un runtime shader compiler renderebbe questa operazione piuttosto tediosa e complicherebbe in maniera significativa il codice stesso della libreria.

3.2 Viewport e Depth Range

Per l'implementazione di viewport e depth range (introdotti nel paragrafo 2.4.2) vi sono due problemi:

- sceGxm presenta una singola funzione per la gestione delle due feature (sceGxmSetViewport) mentre OpenGL ne ha due distinte (glViewport e glDepthRange).
- Gli algoritmi implementativi di sceGxm ed OpenGL sono differenti.

```
1  void glViewport(GLint x,  GLint y,  GLsizei width,  GLsizei height){
2          x_scale = width>>1;
3          x_port = x + x_scale;
4          y_scale = -(height>>1);
5          y_port = DISPLAY_HEIGHT - y + y_scale;
6          sceGxmSetViewport(gxm_context, x_port, x_scale, y_port, \
7              y_scale, z_port, z_scale);
8          gl_viewport.x = x;
9          gl_viewport.y = y;
10         gl_viewport.w = width;
11         gl_viewport.h = height;
12         viewport_mode = 1;
13 }
14
15 void glDepthRange(GLdouble nearVal, GLdouble farVal){
16         z_port = (farVal + nearVal) / 2.0f;
17         z_scale = (farVal - nearVal) / 2.0f;
18         sceGxmSetViewport(gxm_context, x_port, x_scale, y_port, \
19             y_scale, z_port, z_scale);
20         viewport_mode = 1;
21 }
```

Codice 4: Implementazione di glViewport e glDepthRange in VitaGL.

Come mostrato nel cod. 4, il primo punto è facilmente risolvibile facendo in modo che quando una delle due funzioni di OpenGL venga chiamata, vengano alterati i valori della macchina a stati ed aggiornato il viewport di sceGxm (righe 6 e 18). Così facendo, anche se solo il depth range è stato aggiornato, il viewport di sceGxm verrà aggiornato con nuovi valori relativi all'asse Z del viewport mentre verranno lasciati intatti i valori relativi all'asse X ed Y (i quali possono essere aggiornati con l'utilizzo di glViewport).

Per il secondo punto invece, quando glViewport o glDepthRange vengono chiamate, VitaGL effettua una traduzione dei valori in base allo stato aggiornato della macchina in modo da calcolare i valori adeguati relativi all'algoritmo di sceGxm (righe 2-5 e 16-17). Le differenze negli algoritmi sono dati dal fatto che OpenGL considera come origine (x = 0, y = 0) il punto in basso a sinistra della finestra definita dal viewport stesso mentre sceGxm considera come origine il centro della finestra; inoltre OpenGL si aspetta di ricevere come dimensioni (larghezza ed altezza) le dimensioni complete in pixel della finestra mentre sceGxm si aspetta la metà di queste dimensioni in quanto "espande" le dimensioni date ad entrambi i lati rispetto al punto d'origine settato al centro della finestra.

3.3 Blending

OpenGL, per la gestione del blending, fornisce alcune funzioni che modificano lo stato della macchina ed applicano particolari effetti durante il rendering a seconda di esso. In sceGxm, però, tali funzioni non esistono tuttavia è possibile effettuare blending ma questo è legato agli shader. Tuttavia, come abbiamo visto precedentemente, gli shader possono essere utilizzati in sceGxm solamente in forma di binari pre-compilati. In questa situazione sceGxmShaderPatcher ci viene in aiuto fornendoci la possibilità di modificare alcuni settaggi di uno shader già compilato e generare un programma "patchato" da utilizzare per le nostre operazioni di rendering.

```
1  static void _change_blend_factor(SceGxmBlendInfo* blend_info){
2          changeCustomShadersBlend(blend_info);
3
4          sceGxmShaderPatcherCreateFragmentProgram(gxm_shader_patcher,
5                  rgba_fragment_id,
6                  SCE_GXM_OUTPUT_REGISTER_FORMAT_UCHAR4,
7                  msaa_mode,
8                  blend_info,
9                  NULL,
10                 &rgba_fragment_program_patched);
11
12         sceGxmShaderPatcherCreateFragmentProgram(gxm_shader_patcher,
13                 texture2d_fragment_id,
14                 SCE_GXM_OUTPUT_REGISTER_FORMAT_UCHAR4,
15                 msaa_mode,
16                 blend_info,
17                 NULL,
18                 &texture2d_fragment_program_patched);
19
20         sceGxmShaderPatcherCreateFragmentProgram(gxm_shader_patcher,
21                 texture2d_rgba_fragment_id,
22                 SCE_GXM_OUTPUT_REGISTER_FORMAT_UCHAR4,
23                 msaa_mode,
24                 blend_info,
25                 NULL,
26                 &texture2d_rgba_fragment_program_patched);
27
28 }
29
30 void change_blend_factor(){
31         static SceGxmBlendInfo blend_info;
32         blend_info.colorMask = blend_color_mask;
33         blend_info.colorFunc = blend_func_rgb;
34         blend_info.alphaFunc = blend_func_a;
35         blend_info.colorSrc = blend_sfactor_rgb;
36         blend_info.colorDst = blend_dfactor_rgb;
37         blend_info.alphaSrc = blend_sfactor_a;
38         blend_info.alphaDst = blend_dfactor_a;
39
```

```
40          _change_blend_factor(&blend_info);
41          cur_blend_info_ptr = &blend_info;
42          if (cur_program != 0){
43                  reloadCustomShader();
44          }
45  }
```

Codice 5: Gestione del blending in VitaGL.

Da come mostrato nel cod. 5, ogniqualvolta una delle funzioni di OpenGL atte a modificare i fattori di blending è chiamata, VitaGL aggiorna lo stato della macchina relativo al blending (struttura **blend_info** nelle righe 31-38) e viene eseguita una creazione di nuovi programmi a partire dagli shader originali utilizzando i nuovi settaggi di blending (righe 4-26). I programmi generati sostituiranno quelli in utilizzo in precedenza (es. **rgba_fragment_program_patched** nella riga 10). Questo risulta possibile poichè sceGxmShaderPatcher implementa un sistema di garbage collection basato su reference counting in modo che i programmi non in utilizzo da molto tempo vengano automaticamente rimossi dalla memoria ed inoltre permette a questa creazione di programmi sistematica di essere efficente in quanto, se un programma con gli stessi settaggi è stato generato recentemente, verrà riutilizzato quest'ultimo anzichè effettuare una nuova generazione.

3.4 vglDrawObjects e vglMapHeapMem

Mediante i paragrafi precedenti è stato spiegato come VitaGL ha potuto implementare le feature richieste da vitaQuakeIII. Tuttavia per ovviare a grossi problemi di performance dovuti a come sceGxm si aspetta di ricevere i dati durante una chiamata di stampa (sceGxmDraw), è stata implementata una nuova funzione unica in VitaGL, compatibile con tutte le funzioni originali di OpenGL implementate, che permette di effettuare una chiamata di stampa a schermo in maniera molto più efficente delle funzioni convenzionali previste da OpenGL: **vglDrawObjects**.

I due grossi bottleneck che le funzioni di stampa previste da OpenGL presentano una volta implementate in VitaGL sono dovuto dal fatto che vertici, coordinate delle texture e valori dei colori devono essere obbligatoriamente posizionati su memoria accessibile alla GPU inoltre, per via del fatto che sceGxm non permette la modifica a runtime del codice degli shader utilizzati per l'implementazione della fixed function pipeline, questi ultimi avranno un ordine stabilito per il passaggio di stream di dati (vertici, coordinate delle texture e valori RGB/RGBA da utilizzare per il blending di ogni vertice) che non sempre rispetta l'ordine nel quale le originali funzioni di OpenGL li utilizzano.

Il primo problema è stato risolto tramite l'implementazione della funzione **vglMapHeapMem**: quest'ultima, una volta chiamata, mappa l'intero heap fornito da newlib sulla GPU con attributi di scrittura e lettura. Così facendo, memoria semplicemente allocata tramite malloc sarà direttamente utilizzabile da VitaGL, senza richiedere un processo extra di spostamento da RAM a VRAM (quest'ultima infatti sarà sempre mappata ed accessibile dalla GPU). Il secondo problema è stato risolto mediante l'introduzione di funzioni correlate a vglDrawObjects che permettono di fornire a quest'ultima funzione dei puntatori a zone di memoria già precedentemente mappate per la GPU in modo che vglDrawObjects non debba effettuare nessun operazione di riordinamento dei dati da fornire come stream agli shader. Così facendo l'operazione di riordinamento sarà da effettuare al di fuori della libreria stessa se si decide di utilizzare questa funzione, cosa che può risultare estremamente vantaggiosa a seconda di come i dati vengano forniti originariamente dal programma che si intende adattare.

```
1 // VGL_EXT_gpu_objects_array extension
2 void vglColorPointerMapped(GLenum type, const GLvoid* pointer);
3 void vglDrawObjects(GLenum mode, GLsizei count, GLboolean implicit_wvp);
4 void vglIndexPointerMapped(const GLvoid* pointer);
5 void vglTexCoordPointerMapped(const GLvoid* pointer);
6 void vglVertexPointerMapped(const GLvoid* pointer);
```

Codice 6: Prototipi dell'estensione VGL_EXT_gpu_objects_array di VitaGL.

Come mostrato nel cod. 6, VitaGL presenta un'estensione denominata **VGL_EXT_gpu_objects_array** che espone agli sviluppatori la funzione precedentemente descritta nel dettaglio (vglDrawObjects) e le varie funzioni per l'utilizzo di memoria già mappata ed ordinata (funzioni vgl*PointerMapped).

4 vitaQuakeIII, l'implementazione

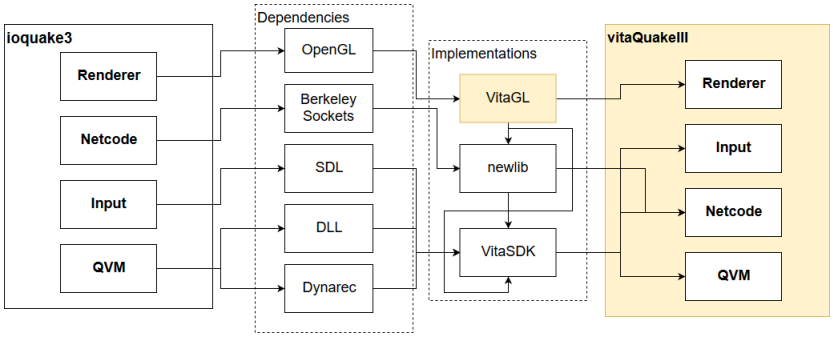

Figura 9: Struttura del lavoro di porting effettuato

Ora che VitaGL, una delle parti fondamentali per l'implementazione di vitaQuakeIII [12], è stata trattata in dettaglio, passiamo ad analizzare come il lavoro di porting di ioquake3 è stato svolto. Nella figura 9, è possibile consultare la struttura originale di ioquake3 e come ogni componente sia stato portato tramite VitaSDK. La prima cosa che è stata analizzata è la lista delle dipendenze (librerie) richieste da ioquake3. Fortunatamente la maggior parte di quest'ultime erano già disponibili su vdpm (introdotto nel paragrafo 2.4) con l'eccezione di alcune librerie opzionali che sono state prontamente disabilitate per la nostra build (es. cURL utilizzato per il download automatico di mappe in caso un giocatore tentasse di connettersi ad una partita in un server online su di una mappa non posseduta dal giocatore). L'unica grossa dipendenza mancante è libGL (OpenGL) che è stata prontamente rimpiazzata da VitaGL. Inoltre, seppure vdpm fornisce una implementazione parziale di SDL2 (Simple DirectMedia Layer, una libreria multiuso che permette la creazione di codice multipiattaforma per quel che riguarda input, codice di rete, rendering, timer, etc...), quest'ultima utilizza per il rendering una libreria non compatibile con VitaGL (vita2d, una libreria esclusivamente per il rendering 2D accelerato sfruttante anch'essa sceGxm) pertanto il backend fornito da SDL2, per quanto concerne l'input e l'output audio, è stato riscritto mediante l'utilizzo di funzioni provenienti dal VitaSDK che tratteremo nello specifico nei paragrafi successivi.

Sebbene vitaQuakeIII sia solamente una modifica del codice di ioquake3 (differentemente da VitaGL che è un lavoro compiuto ex novo), esso è stato realizzato mediante mesi di lavoro in quanto costituisce l'adattazione ed ottimizzazione di un engine di gioco dell'ordine delle centinaia di migliaia di righe di codice.

4.1 Input

Con l'assenza di SDL come backend, l'intero codice di gestione degli input dovrà essere riscritto in modo da utilizzare VitaSDK. Inoltre la mancanza di una tastiera ed un mouse deve essere presa in conto durante l'implementazione del nuovo codice.

La prima cosa che è stata fatta è un'analisi delle funzioni più utilizzate dai giocatori di Quake III: Arena così da decidere quali tasti erano sacrificabili in quanto, come detto in precedenza, la Sony PlayStation® Vita possiede solamente 12 tasti programmabili, due analogici e due touchpad utili al nostro scopo. La scelta ricadde sul seguente schema di tasti:

- Tasti direzionali ed/o analogico sinistro per il movimento del personaggio.
- Analogico destro ed/o touchscreen per il movimento della telecamera.
- Dorsale sinistro per la mira.
- Dorsale destro per il fuoco.
- Tasto Croce per il salto.
- Tasto Quadrato per accucciarsi.
- Tasto Cerchio per utilizzare oggetti ed interagire con l'ambiente.
- Tasto Triangolo per cambiare l'arma in uso.
- Tasto Start per metttere il gioco in pausa.
- Tasto Select per mostrare il punteggio attuale della partita.

Da notare come il giocatore possa comunque personalizzare lo schema dei tasti come meglio desidera nei settaggi di gioco seppure lo schema di default proposto risulta essere una scelta ottimale.

```
1  void Sys_SetKeys(uint32_t keys, int time){
2          if((keys & SCE_CTRL_START) != (oldkeys & SCE_CTRL_START))
3                  Key_Event(K_ESCAPE, (keys & SCE_CTRL_START) == SCE_CTRL_START, time);
4          if((keys & SCE_CTRL_SELECT) != (oldkeys & SCE_CTRL_SELECT))
5                  Key_Event(K_ENTER, (keys & SCE_CTRL_SELECT) == SCE_CTRL_SELECT, time);
6
7      ...
8  }
9
10 void IN_Frame( void )
11 {
12         SceCtrlData keys;
13         sceCtrlPeekBufferPositive(0, &keys, 1);
14         int time = Sys_Milliseconds();
15         if(keys.buttons != oldkeys)
16                 Sys_SetKeys(keys.buttons, time);
17         oldkeys = keys.buttons;
18
19     ...
20 }
```

Codice 7: Parte del codice relativo alla gestione dei tasti in vitaQuakeIII.

Come mostrato nel cod. 7, per l'implementazione dei tasti standard (tasti programmabili), è stata riscritta la funzione **IN_Frame** (righe 10-20) che è responsabile del polling degli input e relativo cambiamento nello stato dei tasti nel codice di gioco. Il nuovo codice chiamerà la funzione **sceCtrlPeekBufferPositive** (riga 13) che restituisce lo stato attuale (sotto forma di bitmask) dei tasti premuti sulla console. Tale bitmask verrà processata e tradotta nei relativi tasti di gioco in maniera analoga a come funzionava il codice originale per il backend SDL (righe 1-8).

```
1         // Emulating mouse with touch
2         SceTouchData touch;
3         sceTouchPeek(SCE_TOUCH_PORT_FRONT, &touch, 1);
4         if (touch.reportNum > 0){
5                 if (old_x != -1)
6                     Com_QueueEvent(time, SE_MOUSE,
7                         (touch.report[0].x - old_x),
8                         (touch.report[0].y - old_y),
9                         0, NULL);
10                old_x = touch.report[0].x;
11                old_y = touch.report[0].y;
12        }else old_x = -1;
13
14        // Emulating mouse with right analog
15        int right_x = (keys.rx - 127) * 256;
16        int right_y = (keys.ry - 127) * 256;
17        IN_RescaleAnalog(&right_x, &right_y, 7680);
18        hires_x += right_x;
19        hires_y += right_y;
20        if (hires_x != 0 || hires_y != 0) {
21                // increase slowdown variable to slow down aiming,
22                // could be made user-adjustable
23                int slowdown = 1024;
24                Com_QueueEvent(time, SE_MOUSE, hires_x / slowdown,
25                    hires_y / slowdown, 0, NULL);
26                hires_x %= slowdown;
27                hires_y %= slowdown;
28        }
```

Codice 8: Emulazione del mouse in vitaQuakeIII.

Come mostrato nel cod. 8, per l'implementazione della camera, l'analogico destro ed il touchscreen sono stati programmati per emulare il mouse in quanto quest'ultimo, nel codice originale, è utilizzato per il movimento del cursore nei menù di gioco e per il movimento della telecamera durante le partite. I dati dell'analogico destro, come mostrato nelle righe 17-28, non vengono direttamente processati bensì vengono prima scalati ad una dimensione maggiore in modo da essere coerenti con il range utilizzato dal touchscreen (il touchscreen ha un range che va da 0 a 1980 per l'asse X e da 0 a 1088 per l'asse Y mentre i range degli analogici vanno da 0 a 255 per entrambi gli assi.).

```
1         // Emulating keys with left analog
2         // (TODO: Replace this dirty hack with a serious implementation)
3         uint32_t virt_buttons = 0x00;
4         if (keys.lx < 80) virt_buttons += LANALOG_LEFT;
5         else if (keys.lx > 160) virt_buttons += LANALOG_RIGHT;
6         if (keys.ly < 80) virt_buttons += LANALOG_UP;
7         else if (keys.ly > 160) virt_buttons += LANALOG_DOWN;
8         if (virt_buttons != oldanalogs){
9                 if((virt_buttons & LANALOG_LEFT) != (oldanalogs & LANALOG_LEFT))
10                        Key_Event(K_AUX7,
11                            (virt_buttons & LANALOG_LEFT) == LANALOG_LEFT, time);
12                if((virt_buttons & LANALOG_RIGHT) != (oldanalogs & LANALOG_RIGHT))
13                        Key_Event(K_AUX8,
14                            (virt_buttons & LANALOG_RIGHT) == LANALOG_RIGHT, time);
15                if((virt_buttons & LANALOG_UP) != (oldanalogs & LANALOG_UP))
16                        Key_Event(K_AUX9,
17                            (virt_buttons & LANALOG_UP) == LANALOG_UP, time);
18                if((virt_buttons & LANALOG_DOWN) != (oldanalogs & LANALOG_DOWN))
19                        Key_Event(K_AUX10,
20                            (virt_buttons & LANALOG_DOWN) == LANALOG_DOWN, time);
21        }
22        oldanalogs = virt_buttons;
```

Codice 9: Utilizzo dell'analogico sinistro in vitaQuakeIII.

Infine, come mostrato nel cod. 9, per l'implementazione dell'analogico sinistro, i dati di quest'ultimo vengono utilizzati per emulare quattro tasti fisici (righe 4-7) in base alla posizione dell'analogico (sopra, sotto, destra, sinistra). Così facendo si perde in precisione (il giocatore non potrà muovere leggermente l'analogico per permettere al personaggio di camminare lentamente ad esempio) ma l'implementazione risulta essere molto più semplice e non intacca in maniera eccessiva le performance dell'engine.

4.2 Netcode

VitaSDK fornisce un'implementazione parziale dei socket Berkeley [10]. Tuttavia questa implementazione manca di alcune feature richieste da ioquake3 per il corretto funzionamento del multiplayer online. Per ovviare al problema, le parti mancanti sono state riscritte utilizzando le funzioni fornite da VitaSDK (modulo SceNet). Principalmente, due funzionalità sono state riscritte seguendo questo criterio: la parte di codice utilizzata per il rilevamento dell'indirizzo IP locale della console ed il codice che si occupa della conversione di un IP dal formato binario (all'interno di una struct sockaddr) in formato testuale.

Nel primo caso, VitaSDK fornisce una comoda funzione che effettua esattamente ciò di cui abbiamo bisogno (basta eseguire una chiamata alla funzione **sceNetCtlInetGetInfo**, pensata per ottenere informazioni sulla rete locale in uso, fornendo come primo parametro il valore **SCE_NETCTL_INFO_GET_IP_ADDRESS**).

Nel secondo caso, la funzione originale (utilizzante **getnameinfo** non disponibile con VitaSDK) è stata riscritta in modo da utilizzare le funzioni di SceNet equivalenti ad **ntohl** ed **ntohs** nei sistemi Unix come mostrato nel cod. 10 sottostante.

```
1 static void Sys_SockaddrToString(char *dest, int destlen, struct sockaddr *input){
2         int haddr = sceNetNtohl(((struct sockaddr_in *)input)->sin_addr.s_addr);
3 
4         sprintf(dest, "%d.%d.%d.%d:%d",
5             (haddr >> 24) & 0xff, (haddr >> 16) & 0xff,
6             (haddr >> 8) & 0xff, haddr & 0xff,
7             sceNetNtohs(((struct sockaddr_in *)input)->sin_port));
8 
9 }
```

Codice 10: Conversione di IP da binario a testuale in vitaQuakeIII.

4.3 Renderer

ioquake3 presenta due renderer dalle finalità completamente diverse:

- Un renderer basato su OpenGL 1.1: questo è la continuazione del renderer originale dell'id Tech 3. Esso è finalizzato al rendere i giochi commerciali sviluppati utilizzando l'originale codice sorgente di id Tech 3 il più fedeli possibili in termini di effetti grafici.
- Un renderer basato su OpenGL 2.0: questo è un nuovo renderer sviluppato dai programmatori di ioquake3 stessi. Esso rimpiazza il sistema di shading fornito da id Tech 3 con un reale utilizzo di shaders che vengono compilati a runtime a seconda degli shader (nel formato previsto da id Tech 3) utilizzati dai giochi. Esso è finalizzato ad espandere l'originale engine e renderlo più potente introducendo nuove feature e migliorando quelle esistenti.

Per via dell'assenza di un runtime shader compiler, la scelta per vitaQuakeIII ricadde sul renderer basato su OpenGL 1.1. Inoltre quest'ultimo richiede meno risorse per l'utilizzo per via dell'assenza di diverse nuove feature presenti nell'altro renderer pertanto, essendo Sony PlayStation® Vita una console handheld piuttosto limitata in fatto di potenza di calcolo e memoria disponibile, risulta essere la scelta migliore nel nostro caso.

Con la creazione di VitaGL, moltissimo codice è stato utilizzabile fin dal principio senza alcuna modifica, tuttavia il codice relativo alle chiamate di stampa è stato completamente riscritto per utilizzare **vglDrawObjects** in modo da migliorare sensibilmente le performance dell'engine. La prima operazione effettuata è stata rimpiazzare il codice di inizializzazione della libreria aggiungendo il codice unico di VitaGL atto ad inizializzare la macchina a stati di sceGxm.

```
1 if (!inited){
2                 vglInitExtended(0x100000, glConfig.vidWidth, glConfig.vidHeight,
3                     0x1000000, SCE_GXM_MULTISAMPLE_4X);
4                 vglUseVram(GL_TRUE);
5                 vglMapHeapMem();
6                 inited = 1;
7                 cur_width = glConfig.vidWidth;
8 }
```

Codice 11: Inizializzazione di VitaGL in vitaQuakeIII.

Da notare nel cod. 11 come sia possibile con sceGxm utilizzare MSAA (Multi Sampling Anti Aliasing che, da come suggerisce il nome, è una tecnica che sfrutta l'utilizzo di sampling più grandi, ad esempio samples di grandezza 2x2 per frammento nel caso dell'MSAA 4x, e di un successivo downscale in modo da ridurre l'effetto di aliasing, ossia scalettamento, durante il rendering 3D) senza intaccare le performance del renderer poichè le operazioni necessarie sono effettuate "on chip" dalla GPU stessa pertanto la CPU (vero bottleneck della console) non è minimamente intaccata dal suo utilizzo.

Figura 10: Antialiasing comparison con VitaGL (Zoom 300%)

Subito dopo il codice di inizializzazione di VitaGL, è stato aggiunto del codice atto a preparare l'utilizzo estensivo di vglDrawObjects.

```
1        int i;
2        indices = (uint16_t*)malloc(sizeof(uint16_t*)*MAX_INDICES);
3        for (i=0;i<MAX_INDICES;i++){
4                indices[i] = i;
5        }
6        vglIndexPointerMapped(indices);
7        glEnableClientState(GL_VERTEX_ARRAY);
8        gVertexBufferPtr = (float*)malloc(0x400000);
9        gColorBufferPtr = (uint8_t*)malloc(0x200000);
10       gTexCoordBufferPtr = (float*)malloc(0x200000);
11       gColorBuffer255 = (uint8_t*)malloc(0x3000);
12       memset(gColorBuffer255, 0xFF, 0x3000);
13       gVertexBuffer = gVertexBufferPtr;
14       gColorBuffer = gColorBufferPtr;
15       gTexCoordBuffer = gTexCoordBufferPtr;
```

Codice 12: Preparazione dei buffer utilizzati da vglDrawObjects.

Il cod. 12 mostrato qui sopra evidenzia la preparazione di buffer che verranno utilizzati come delle pool di memoria dal renderer: innanzitutto vengono allocati degli indici ed inizializzati in maniera consecutiva poichè il codice di rendering verrà riscritto in modo che gli indici saranno sempre progressivi permettendoci così di non dover riallocare ad ogni singola call di stampa dei nuovi indici riducendo così il carico di lavoro della CPU. Dopodichè, sono stati allocati in memoria quattro differenti buffer (gVertexBufferPtr, gColorBufferPtr, gTexCoordBufferPtr e gColorBuffer255). Da come suggeriscono i nomi delle variabili, questi sono puntatori a buffer utilizzati per vertici, colori RGB e coordinate delle texture utilizzati durante le call di stampa. Menzione particolare per

gColorBuffer255 che viene istantaneamente inizializzato con valori fissi di 255 (0xFF) per tutta la grandezza del buffer: questo buffer è utilizzato quando id Tech 3 effettua una call di stampa con colore fisso 0xFFFFFFFF (ossia colore bianco completamente opaco). Così facendo possiamo evitare un grosso carico di lavoro sulla CPU poichè eviteremo di far allocare un valore fisso su di gColorBufferPtr per un numero arbitrariamente grande di volte bensì forniremo alla GPU direttamente l'indirizzo in memoria di gColorBuffer255 in quanto quest'ultimo risulta essere già mappato sulla GPU grazie alla call di vglMapHeapMem nel cod. 11.

```
1 float *pPos = gVertexBuffer;
2 for ( i = 0; i <= NUM_BEAM_SEGS; i++ ) {
3         memcpy(gVertexBuffer, start_points[ i % NUM_BEAM_SEGS], sizeof(vec3_t));
4         gVertexBuffer+=3;
5         memcpy(gVertexBuffer, end_points[ i % NUM_BEAM_SEGS], sizeof(vec3_t));
6         gVertexBuffer+=3;
7 }
8 vglVertexPointerMapped(pPos);
9 vglDrawObjects(GL_TRIANGLE_STRIP, (NUM_BEAM_SEGS + 1) * 2, GL_TRUE);
```

Codice 13: Esempio di funzione di stampa in vitaQuakeIII.

Infine, come mostrato nell'esempio del cod. 13, tutte le chiamate di stampa sono state rimpiazzate con un codice che, se necessario, prima si occupa di riordinare i dati dello stream a seconda degli indici e dopo effettua una call a vglDrawObjects bypassando, in tal modo, il codice di VitaGL atto ad effettuare la medesima operazione ma in maniera meno efficiente (in quanto le funzioni standard di stampa dovrebbero anche calcolare l'indice massimo utilizzato prima di effettuare l'ordinamento).

C'è da precisare che il rendering dei portali e degli specchi sono stati disabilitati in modo che mostrino delle texture nere opache al loro posto. Questo perchè la loro implementazione richiede l'utilizzo di clip planes (Piani aggiuntivi a quelli definiti dalla viewport per nascondere elementi che risultano essere al di fuori di questi piani stessi) e VitaGL non supporta tale feature. Tuttavia essendo feature secondarie, l'assenza di essi risulta irrilevante per il corretto utilizzo dell'engine.

4.4 QVM

vitaQuakeIII implementa le tre modalità di utilizzo delle QVM analizzate nel dettaglio nel paragrafo 2.2.1 in maniera completa e selezionabile dall'utente stesso mediante la modifica del file di configurazione dell'engine (seppure sia caldamente consigliato l'utilizzo delle librerie dinamiche).

Il primo metodo, ossia l'interprete bytecode puro, è costituito da codice totalmente multipiattaforma pertanto esso risulta essere utilizzabile per il nostro progetto fin da subito senza alcuna modifica.

Il secondo metodo, ossia il dynarec, è formato da diversi codici da abilitare durante la fase di compilazione tramite istruzioni al preprocessore del compilatore a seconda dell'architettura della macchina target. Il codice originale di id Tech 3 prevedeva tale codice solamente per macchine x86 ed x64; tuttavia gli sviluppatori di ioquake3 introdussero nuove versioni di tali ricompilatori dinamici per altre architetture. Fortunatamente l'architettura della CPU utilizzata da Sony PlayStation® Vita (ossia ARMv7) è supportata da ioquake3 pertanto con lievi modifiche al codice di allocazione della memoria per l'utilizzo del dynarec (necessario in quanto VitaSDK supporta il dynarec ma richiede che il buffer di memoria utilizzato per quest'ultima sia allineato di 1 megabyte ed utilizzando la funzione **sceKernelAllocMemBlockForVM**), questo metodo di utilizzo delle QVM è stato implementato correttamente. Ciononostante, tale metodo presenta diversi problemi di stabilità (ad esempio al termine di una partita, a seconda dello stato della QVM **game**, possono presentarsi crash dovuti a segmentation fault).

Il terzo metodo, ideato per ovviare ai problemi di stabilità presentati dal secondo metodo ed ai problemi di performance presentati dal primo, è il più complesso da implementare ma anche la scelta migliore per il nostro progetto. VitaSDK permette la creazione di moduli (in formato .suprx) che possono essere caricati dinamicamente e lanciati mediante la funzione **sceKernelLoadStartModule** e quest'ultimi possono esporre funzioni da esportare tramite un database yaml che viene processato da VitaSDK durante la compilazione; pertanto, similarmente a ciò che accade con i file DLL su sistemi Windows, VitaSDK permette l'utilizzo di librerie dinamiche seppure con un inconveniente: tali librerie non condividono la memoria in uso pertanto ogni QVM compilata in questo modo non potrà utilizzare la memoria dinamica disponibile all'eseguibile principale, nemmeno se tali funzioni vengono importate da quest'ultimo. Per ovviare al problema, è stato implementato, in ogni QVM, un heap tramite newlib molto ristretto (solamente 2 MB) così che le funzioni utilizzate all'interno delle librerie stesse funzionassero in maniera corretta e, allo stesso tempo, la libreria non utilizzasse troppa memoria rendendo l'eseguibile principale troppo limitato nelle risorse disponibili.

5 Conclusioni

Figura 11: Comparison tra ioquake3 su PC (sinistra) e vitaQuakeIII (destra)

5.1 Sviluppi futuri

In futuro, sia VitaGL che vitaQuakeIII potranno vedere ulteriori sviluppi; per quanto riguarda VitaGL, l'implementazione delle clip planes permetterebbe di abilitare il rendering di portali e specchi in vitaQuakeIII. Inoltre un grosso lavoro di code refactoring è tutt'ora in corso e procederà nei prossimi mesi per rendere il codice più leggibile, commentato ed ottimizzato in quanto, durante le prime fasi di sviluppo, la complessità ed il costo degli algoritmi utilizzati sono stati presi in considerazione solo in minima parte.

Parlando di vitaQuakeIII invece, il lavoro svolto potrà essere usato per portare in maniera relativamente semplice altre versioni modificate dell'engine id Tech 3 (ve ne sono diverse come ad esempio iortcw, versione modifica di ioquake3 che supporta il titolo Return to Castle Wolfenstein, che ho personalmente trasposto su Sony PlayStation® Vita recentemente in maniera sperimentale utilizzando appunto il codice di vitaQuakeIII come base). Inoltre un lavoro di ottimizzazione del sistema di shading potrebbe portare ad un sensibile miglioramento delle performance dell'engine che tutt'oggi si attesta sui 30-50 frame al secondo. Sono in corso alcuni esperimenti per tentare di rendere il sistema di shading compatibile in maniera nativa con VitaGL (ossia rimpiazzare l'utilizzo di memoria dello stack con memoria dell'heap e fare in modo che il sistema stesso utilizzi una pool di memoria mappata sulla GPU al fine di evitare completamente l'operazione di riordinamento dei dati di stream ed allegerire enormemente il carico di lavoro della CPU) e si spera che tali esperimenti possano condurre ad un reale miglioramento del codebase del progetto.

5.2 Conclusioni personali

Sono completamente soddisfatto dei risultati raggiunti finora da vitaQuakeIII e VitaGL. Entrambi i progetti son stati delle grandissime sfide che mi hanno portato a comprendere in maniera molto più chiara come OpenGL funzioni e come lavorare con codice complesso scritto in maniera professionale da vere e proprie software house come la id Software. Personalmente, continuerò a lavorare ad entrambi i progetti essendo quest'ultimi tutt'ora non completi e spero che gli utenti di Sony PlayStation® Vita continuino a gradire il lavoro svolto come hanno dimostrato finora con i loro 14.000 download di vitaQuakeIII totalizzati in circa un anno.

6 Bibliografia

Riferimenti bibliografici

[1] Cg language specifications. https://developer.download.nvidia.com/cg/Cg_language.html.

[2] Opengl es 1.1 reference pages. https://www.khronos.org/registry/OpenGL-Refpages/es1.1/.

[3] ioquake3 free software first person shooter engine based on the quake 3: Arena and quake 3: Team arena source code. https://ioquake3.org/. Accessed: 2019-04.

[4] Paul Jaquays and Brian Hook. Quake 3 arena shader manual. http://fabiensanglard.net/fd_proxy/quake3/Q3%20Shaders.pdf, 1999.

[5] Yifan Lu. VitaSDK specifications ps vita open sdk specification. https://wiki.henkaku.xyz/vita/images/a/a2/Vita_SDK_specifications.pdf. Accessed: 2019-04.

[6] newlib a conglomeration of several library parts. https://sourceware.org/newlib/. Accessed: 2019-04.

[7] Opengl overview. https://www.opengl.org/about/.

[8] PSVITA tech specs pch-2000 series product specification. https://www.playstation.com/en-us/explore/psvita/system/system-specs/. Accessed: 2019-06.

[9] Quake 3 Source Code Review. http://fabiensanglard.net/quake3/. Accessed: 2019-04.

[10] Berkeley Sockets microchip developer guide. https://microchipdeveloper.com/tcpip:berkeley-sockets. Accessed: 2019-06.

[11] VitaGL opengl wrapper for psvita. https://github.com/Rinnegatamante/vitaGL. Accessed: 2019-04.

[12] vitaQuakeIII ioquake3 port for psvita. https://github.com/Rinnegatamante/vitaQuakeIII. Accessed: 2019-04.

[13] VitaSDK the unofficial sdk for ps vita. https://vitasdk.org/. Accessed: 2019-04.

[14] YAML official yaml web site. https://yaml.org/. Accessed: 2019-06.

Ringraziamenti

Vorrei ringraziare il Professor Lanese per avermi dato la possibilità di svolgere una tesi su di un lavoro che ho effettuato come hobby durante il mio tempo libero unificando due delle mie più grandi passioni: programmazione e videogiochi.

Durante lo sviluppo ho incontrato diverse difficoltà dettate dalla scarsa documentazione disponibile a sviluppatori amatoriali per quel che riguarda SceGxm e dalle mie lacune in materia ma, mediante studi, reverse engineering ed un pizzico di aiuti da altri programmatori, posso ritenermi soddisfatto dello stato attuale del progetto e di ciò che potrà diventare in futuro.

Vorrei infine ringraziare tutti gli sviluppatori al lavoro su di ioquake3, tutti gli sviluppatori che hanno permesso ad un toolchain unico e meraviglioso come VitaSDK di esistere ed, in particolar modo, Francisco José García García (nickname frangarcj sul web) e Sergi Granell (nickname xerpi sul web) per i preziosi consigli fornitimi durante la creazione di VitaGL.