Patrik Jylhä

3D-Renderöinti Vulkan-rajapinnalla

Tietotekniikan kandidaatintutkielma

21. toukokuuta 2023

Jyväskylän yliopisto

Informaatioteknologian tiedekunta

Tekijä: Patrik Jylhä

Yhteystiedot: patrik.e.j.jylha@student.jyu.fi

Ohjaaja: Tuomo Rossi

Työn nimi: 3D-Renderöinti Vulkan-rajapinnalla

Title in English: 3D-Rendering with Vulkan-API

Työ: Kandidaatintutkielma

Opintosuunta: Tietotekniikka

Sivumäärä: 23+0

Tiivistelmä: Nykypäivänä 3D-grafiikkaa käytetään lukuisissa käyttökohteissa, useilla eri käyttöjärjestelmillä ja sitäkin moninaisimmilla laitteilla. Tästä laitteiden ja ohjelmistojen kirjosta johtuen useimmat kehittäjät päätyvät käyttämään alustariippumatonta grafiikkarajapintaa renderöinnin toteutukseen. Tässä tutkielmassa pyritään selvittämään Vulkanin ja OpenGL-rajapinnan eroja sekä tutkitaan syitä miksi kehittäjä valitsisi Vulkanin modernin renderöijän kehitykseen OpenGL:n sijasta.

Avainsanat: Vulkan, OpenGL, GPU, CPU, Grafiikkaliukuhihna, Renderöinti, Alustariippumaton kehitys

Abstract: Today, 3D graphics are use in many different applications, on many different operating systems and on many different devices. Because of this variety of devices and software, most developers end up using a cross-platform graphics API for rendering. In this thesis, we try to find out the differences between Vulkan and OpenGL APIs and to investigate the reasons why a developer would choose Vulkan over OpenGL for the development of a modern renderer.

Keywords: Vulkan, OpenGL, GPU, CPU, Graphics Pipeline, Rendering, Cross-platform development

Termiluettelo

Vulkan	Khronos Groupin kehittämä matalantason alustariippumaton grafiikkaohjelmointirajapinta
OpenGL	Silicon Graphics yhtiön kehittämä korkeantason alustariippumaton grafiikkaohjelmointirajapinta
Varjostin	Ohjelma, joka suoritetaan näytönohjaimella, yleensä grafiikan piirtämiseen tai laskentaan.
Renderöinti	Prosessi, jossa 3D-mallista luodaan 2D-kuva.
Grafiikkaliukuhihna	Koko renderöinti prosessi, aina ohjelman päivityksestä kuvan piirtämiseen asti.
Puskuri	Muistialue, johon dataa voidaan kirjoittaa ja josta sitä voidaan lukea.
Grafiikkarajapinta	Ohjelmointirajapinta, joka mahdollistaa grafiikan piirtämisen näytönohjaimella.

Kuviot

Sisällys

1 Johdanto

Tutkielman tarkoituksena on selvittää, miten reaaliaikainen 3D-renderöinti toteutetaan modernissa Vulkan pohjaisessa applikaatiossa ja varmistua siitä, että Vulkan on todellakin useissa käyttökohteissa parempi vaihtoehto kuin OpenGL. Ensimmäinen versio Vulkanista julkaistiin 2016 ("Vulkan Release" 2016) ja se on nopeasti noussut suosituksi grafiikkarajapinnaksi kehittäjien keskuudessa, jotka haluavat julkaista tehokkaita 3D renderöintiohjelmistoja useammilla alustoilla, käyttämättä alustariippuvaisia grafiikkarajapintoja (Lujan ym. 2019). Historiallisesti OpenGL on ollut suosituin grafiikkarajapinta alustariippumattomien 3D-renderöintisovellusten toteutukseen, mutta nykypäivänä useat kehittäjät suosivat Vulkania, sillä se antaa kehittäjälle enemmän optimointimahdollisuuksia, sillä Vulkan siirtää implementaation yksityiskohtia grafiikka-ajurin puolelta applikaatiopuolelle (Ioannidis ja Boutsi 2020). Vulkan sisältää monia parannuksia OpenGL-rajapintaan verrattuna, kuten moniydintuen, vähennetyn kuormituksen ajuritasolla ja sen, että Vulkanin suunnittelumallin ydin on tarjota ohjelmistokehittäjälle enemmän työkaluja hallita grafiikkaliukuhihnaa ja näytönohjaimen muistia. OpenGL on jo yli kolmekymmentä vuotta vanha ja Khronos Group on käyttänyt kaiken OpenGL kehityksen aikana opitun Vulkanin kehitykseen. Näin Vulkanista on saatu kehitettyä nykyaikainen ja ennen kaikkea tehokas grafiikkarajapinta, joka tarjoaa samat ominaisuudet niin työpöytä- kuin mobiili alustoilla ("Vulkan Wikipedia" 2023). Tämä yhtenäisyys alustojen välillä tekee Vulkanista helppokäyttöisen ja virtaviivaistaa Vulkan pohjaisten sovellusten kehitystä, sekä niiden ylläpitoa.

2 Vulkan kehitys

Vulkania voidaan pitää OpenGL-rajapinnan seuraajana. Vulkan tarjoaa useita uusia ominaisuuksia, joita OpenGL joko ei tarjoa tai piilottaa applikaatio-ohjelmoijan ulottumattomiin. Vulkan pyrkii tarjoamaan ohjelmoijalle mahdollisuuden ottaa mahdollisimman paljon irti käytettävän laitteen suoritustehosta. Kun OpenGL-rajapintaa suunniteltiin, näytönohjaimet olivat enemmän kokoelma useita integroituja prosessoreita kuin erillisiä kortteja mitä ne ovat nykypäivänä (Akeley 1993). Täten OpenGL-rajapinnan arkkitehtuuri mallintaa sen aikaisten prosessorien arkkitehtuuria. OpenGL ei tue moniydinprosessointia, sillä siihen aikaan kaikki mikroprosessorit olivat yksi ytimisiä, niitä oli vain useita samassa työasemassa ("SGI Indigo Workstations" 1998). Vulkanin kehitys alkoi vuonna 2014 ja Khronos Group julkisti Vulkan-rajapinnan 2015 GDC:sä ("Vulkan Accouncement" 2015). Virallinen 1.0 versio julkaistiin 2016, avoimen ohjelmistokehityspaketin kanssa. Uusin versio Vulkanista on 1.3.x. Khronos Group päivittää Vulkania n. kuukauden välein ja kehitys on avoimesti seurattavissa Khronos Groupin GitHub sivulla ("Khronos Group GitHub" 2023). Useat yhtiöt ovat myös kehittäneet kirjastoja ja tekniikoita joilla kehittäjät voivat helpommin siirtyä käyttämään Vulkania jopa vanhoissa ohjelmistoissa, joissa ennen on käytetty OpenGL-rajapintaa ("Transitioning from OpenGL to Vulkan" 2016).

3 Vulkan ohjelmointi

Vulkan on enemmän näytönohjainrajapinta kuin perinteinen grafiikkarajapinta kuten OpenGL. Ohjelmoija voi halutessaan käyttää rajapintaa, vaikka ainoastaan näytönohjaimella laskemiseen ilman minkäänlaista graafista ulostuloa. Grafiikkaohjelmointi on alustavasti monimutkaisempaa Vulkanissa verrattuna OpenGL-rajapintaan, mutta paljon läpinäkyvämpää ohjelmoijalle. Monet OpenGL:ssä ajuritasolla tapahtuvat, yleensä applikaatio-ohjelmoijalta piilotetut konseptit ja toiminnot ovat Vulkanissa näkyviä, mikä tekee ongelmien havaitsemisesta yleensä helpompaa. Rajapinta koostuu Vulkan objekteista, jotka on nimetty "Vk-etuliitteellä, kuten VkInstance, sekä Vulkan funktioista jotka on nimetty "vk-etuliitteellä kuten vkCreateInstance(). Rajapintaa käytetään yleensä täyttämällä ensin jonkinlainen Vulkan objekti ja sen jälkeen kutsumalla jotain Vulkan funktiota tällä kyseisellä objektilla. Vulkan objekteja ei tule tulkita osoittimiksi eikä numeroiksi, oikeastaan ohjelmoijan ei tule tulkita Vulkan objekteja mitenkään, vaan ainoastaan viedä niitä rajapinnan dokumentaation mukaisesti viitteinä. Objektit tulee luonnollisesti myös tuhota, kun niitä ei enää tarvita.

Esitellään nyt yleisesti Vulkan objektit ja niiden yleistä käyttöä. Käymme myös läpi pääpiirteittäin Vulkan-rajapinnan ominaisuuksia ja yleisesti eroja OpenGL-rajapinnan kanssa.

Alaluku käyttää lähteenä Vulkan-rajapinnan dokumentaatiota (“Vulkan® 1.3.246 - A Specification” 2023), Alexander Overvoorde:n Vulkan ohjekirjaa (“Vulkan Tutorial” 2023) sekä Adam Sawicki:n artikkelia Vulkan objekteista (“Understanding Vulkan Objects - Adam Sawicki” 2017) jos lähdettä ei erikseen mainita.

3.1 Perusteet

Vulkanin alustus tapahtuu luomalla instanssiobjekti (VkInstance), joka edustaa linkkiä Vulkan-rajapinnan ja asiakasapplikaation välillä. Instanssi sisältää applikaation keskeisen tilan, jota tarvitaan Vulkan-rajapinnan käyttöön. Tämän takia ohjelmoijan tulisi spesifioida instanssia luodessa kaikki ominaisuudet ja lisäosat, joita ohjelman suorituksen aikana halutaan käyttää. Instanssiobjekti luodaan kutsumalla vkCreateInstance funktiota. Instanssiobjekti on yleisesti ensimmäinen objekti, joka luodaan Vulkan-applikaatiota alustettaessa. Instanssiobjekti on

myös yleensä viimeinen objekti, joka tuhotaan Vulkan-applikaatiota suljettaessa.

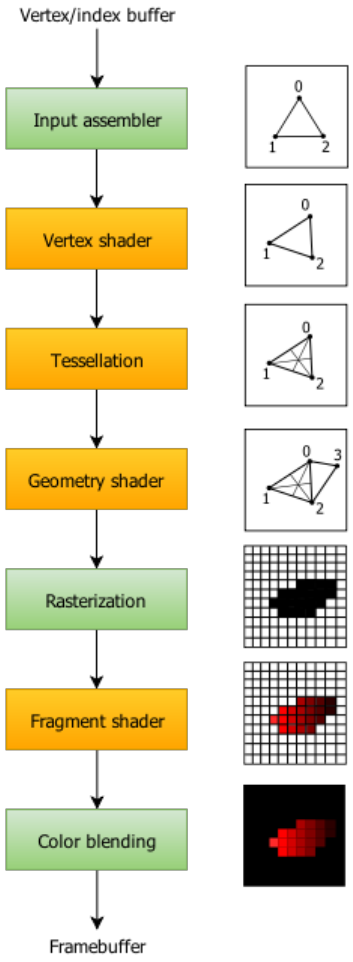
Seuraavana luodaan fyysinen laite (VkPhysicalDevice) objekti, joka edustaa fyysistä laitetta, jolla Vulkan-rajapinta on käytettävissä, toisin sanoen näytönohjainta. Laiteobjekti luodaan kyselemällä instanssiobjektilta tunnistetut laitteet ja niiden ominaisuudet ja mahdolliset rajoitteet. Fyysinen laite voi myös luetella saatavilla olevat jonoperheet, joista grafiikkajono on grafiikkaliukuhihnan kannalta oleellisin, palataan näihin pian.

Fyysinen laite voi luetella muistikeot ja muistityypit niiden sisällä. Muistikeko esittää tässä yhteydessä tiettyä fyysistä muistiosuutta. Se voi esittää järjestelmämuistia, tiettyä osaa videomuistia näytönohjaimessa tai mitä vain muuta alusta- tai laitekohtaista muistia, jota implementaatio haluaa esittää. Ohjelmoijan täytyy aina määritellä muistityyppi varauksen yhteydessä.

Looginen laite edustaa alustettua Vulkania tukevaa laitetta, joka on valmis luomaan kaikki muut tarvittavat objektit. Jos on ohjelmoinut DirectX:llä, tämä muistuttaa sieltä tuttua device-objektia. Kun Vulkanissa luodaan laiteobjektia, tulee spesifioida kaikki ominaisuudet, joita halutaan käyttää. Luonnissa tulee myös esittää kaikki halutut jonot, joita tullaan käyttämään, niiden määrä ja mihin jonoperheisiin ne kuuluvat.

3.2 Grafiikkaliukuhihna

Grafiikkaliukuhihnalla tarkoitetaan tässä luvussa koko piirtoprosessia. Tämä sisältää kaikki vaiheet alkaen komentojonojen luomisesta ja lopettaen näytön päivitykseen. Kuviossa 1 on esitetty tavanomainen Vulkan grafiikkaliukuhihna yhdelle piirtokomennolle. Reaaliaikaisessa sovelluksessa grafiikkaliukuhihna suoritetaan useita kertoja sekunnissa, täten kuvio esittää yhtä kokonaista piirtoprosessia kyseiselle komennolle. Tavanomaisesti yksi komento vastaa aina yhtä mallia nykyisessä kulisissa, mutta sovelluskehittäjä voi vapaasti myös koota useita malleja yhteen komentoon, josta voi mahdollisesti olla suorituskykyetuja. Grafiikkaliukuhihna luodaan instanssiobjektin alustuksen jälkeen ja meidän tulee kuvailla halutut grafiikkaliukuhihnan vaiheet grafiikkaliukuhihnaa alustaessa. Sovelluskehittäjällä on mahdollisuus muuttaa grafiikkaliukuhihnan rakenneta Vulkanissa, joten kuviossa 1 esitetty grafiikkaliukuhihna ei päde jokaiselle Vulkan pohjaiselle sovellukselle. Yleisin tapa muuttaa grafiikkaliukuhihnaa on lisätä varjostimien (shader) määrää. Kehittäjä voi myös täysin vapaasti olla hyödyntämättä haluttuja varjostinvaiheita.



Kuvio 1: Vulkan grafiikkaliukuhihna (github.com/Overv/VulkanTutorial)

Oletetaan, että sovellus on päivittänyt tilansa ja tarjonnut grafiikkaliukuhihnalle kaksi puskuria, joissa pidetään piirrettävien mallien kärkipisteitä, sekä nyt piirrettävien kärkipisteiden indeksejä. Kärkipisteet voivat sovelluskohtaisesti sisältää mitä vain attribuutteja. Sovelluskehittäjän vastuulla on tarjota grafiikkaliukuhihnalle ainoastaan halutut kärkipisteet sekä kaikki muu tarvittava data. Vulkanissa täytyy määritellä syötteen kokoaja (input assembler), jossa Vulkanille kerrotaan, millaisessa formaatissa syöte on, syötteenä viedään kokoelma kolmioiden kärkipisteitä. Syötteen kokoaja vie sen prosessoiman datan kärkipistevarjostimelle (vertex shader).

Kärkipistevarjostin on ohjelma, joka suoritetaan jokaiselle kärkipisteelle tasan kerran, ja sen

tarkoitus on muokata kärkipisteitä halutulla tavalla. Kärkipisteitä muutetaan sovelluksen nykyisen tilan avulla, esimerkiksi peleissä sovelluskerroksella pidetään yllä hahmojen nykyistä paikkaa pelimaailmassa, joten voimme hyödyntää tätä paikkadataa kärkipistevarjostimessa luomaan kolmiulotteiselta tuntuvan kokemuksen loppukäyttäjälle. Halutessaan kehittäjä voi hyödyntää useita varjostinvaiheita grafiikkaliukuhihnassa. Vulkan tarjoaa useita varjostintyyppejä kuten kuviossa 1 esitetyt tessalointi- ja geometriavarjostimet. Tessalloinnilla tarkoitetaan monikulmioiden jakamista pienempiin monikulmioihin, jotta malleille saadaan enemmän resoluutiota. Geometriavarjostimissa kehittäjä voi halutessaan generoida lisää monikulmioita kärkipistepuskurista eroaviin paikkoihin. Näin voidaan ohjelmallisesti esimerkiksi generoida monimutkaisia maastoja videopeleissä.

Rasteroija (rasterizer) muuttaa mallien primitiivit paloiksi (fragment) jotka täyttävät pikselit kuvassa, jota myöhemmin käytetään pikselivarjostimessa (fragment shader). Tässä vaiheessa rasteroija hylkää pikselit, jotka ovat piirrettävän kuvan ulkopuolella sekä useasti hylätään myös toisten kappaleiden takana olevat kappeleiden osat syvyystarkastelun avulla. Pikselivarjostimessa käsitellään jokainen pikseli joka on jäänyt jäljelle rasteroijan hylättyä tarpeettomat pikselit. Pikselivarjostimen avulla malleihin lisätään tekstuuri sekä valaistus käyttäen hyödyksi kärkipistevarjostimessa prosessoitua dataa, kuten kärkipisteiden paikkoja ja niiden muita attribuutteja. Lopulta pikselit kootaan lopulliseksi kuvaksi kaikista renderöidyistä palasista. Palaset voivat joko korvata kokonaan pikseleitä tai tilanteessa, jossa halutaan malleille läpinäkyvyyttä, pikseleitä voidaan yhdistellä tai lisätä toisiinsa. Yhdistetyt pikselit on nyt saatu renderöityä niiden lopullisen muotoon ja tallennettua kehyspuskuriin (framebuffer) josta kuva voidaan piirtää näytölle, graaffisenkäyttöliittymään tai tiedostoon. Useasti kuvaan lisätään vielä ennen lopullista piirtoa efektejä, kuten värikorjauksia, resoluution skaalausta tai nykypäivänä jopa tekoälyn avulla tuotettuja korjauksia. Efekteihin ja korjauksiin yleensä käytetään näytönohjainvalmistajien tarjoamia työkaluja kuten esimerkiksi Nvidian DLSS (Deep learning super sampling) kirjastoa (“Nvidia DLSS” 2023).

3.3 Komentopuskuri

Vulkanissa komentojen suoritukseen käytetään jonoja (vkQueue) ja komentopuskureita (vkCommandBuffer). Jonolla tarkoitetaan Vulkan-kontekstissa objektia, joka edustaa komentojo-

noa, joka tullaan suorittamaan loogisella laitteella (vkDevice). Kaikki todellinen työ mitä näytönohjaimella suoritetaan, varastoidaan komentopuskuriobjekteihin (vkCommandBuffer) ja lähetetään jonoihin käyttäen vkQueueSubmit-funktiota. Jos halutaan käyttää useita jonoja, kuten päägrafiikkajonoa sekä erillistä laskentajonoa, voidaan jonoihin lähettää eri vkCommandBuffer-objektit. Täten ohjelmoija voi suorittaa komentoja asynkronisesti ja tämä voi johtaa suuriin aikasäästöihin. Tämä on yksi tapa, jolla Vulkan mahdollistaa nopeamman renderöinnin kuin OpenGL. Jonojen käyttöön tarvitaan vielä yksi objekti, komentoryhmä. Komentoryhmä on yksinkertainen objekti, jolla varataan komentopuskurit muistista. Komentoryhmä liittyy aina tiettyyn jonoperheeseen ja komentopuskuri varataan tietystä komentoryhmästä. Tämä objekti edustaa lukuisten komentojen puskuria, jotka aiotaan suorittaa loogisella laitteella. Vulkan tarjoaa useita funktioita, joita voidaan kutsua komentopuskurilla, käyttäen "vkCmd-alkuisia funktioita. Näillä spesifioidaan järjestys ja parametrit tehtäville, jotka myöhemmin viedään komentopuskurissa jonolle ja lopulta loogiselle laitteelle suoritettavaksi.

3.4 Resurssit ja muistin hallinta

Muistin hallinta Vulkanissa on hyvin läpinäkyvää applikaatiokehittäjälle. Vulkanissa on kahden tyyppisiä resursseja, jotka ovat kuva ja puskuri. Puskuri on näistä se yksinkertaisempi, se on vain tietorakenne mille tahansa lineaariselle datalle. Kuva sen sijaan, koostuu aivan kuten OpenGL:ssä, yksi-, kaksi– tai kolmiulotteisesta datasta, joka on organisoitu laitekohtaisesti. Yleensä tätä kuvaa kutsutaan tekstuuriksi, ja se onkin monissa muissa grafiikkarajapinnoissa nimetty näin, kuten OpenGL:ssä. Puskurit ja tekstuurit ovat Vulkanissa hyvin monikäyttöisiä, meidän täytyy vain alustuksen yhteydessä tarkentaa kyseisen muistiobjektin käyttötarkoitus. Käyttötarkoituksen määrittely optimoinnin ohella auttaa meitä myös löytämään mahdollisia ongelmia, kun tiettyä resurssia käytetään väärin, sillä Vulkanin validointilisäosa ilmoittaa meille väärinkäytöstä.

Vulkanissa kuvalle ja puskurille voidaan viedä useita alustusparametreja, joilla tarkennetaan millaista, kuvaa tai puskuria ollaan luomassa. Yleisiä parametreja kuvalle ovat esim. pikseliformaatti ja dimensio. Kuvalla on myös useita sisäisiä formaatteja. Varjostimet voivat lukea kuviointialkioita (texel) suoraan kuvista, mutta yleisin tapa Vulkanissa on käyttää sample-

reita tekstuureita käytettäessä. Sampleri-objekti ei ole sidottuna mihinkään yhteen kuvaan Vulkanissa. Se on vain joukko parametreja, kuten suodatustila (filtering mode) tai osoitetila (adressing mode) joilla kuvaa voidaan tulkita.

Muistin varaus ei tapahdu kuva- tai puskuriobjekteille luontivaiheessa, vaan varaus on kolmivaiheinen ja täysin ohjelmoijan vastuulla. Puskureita alustettaessa käytetään laitemuistiobjekteja (VkDeviceMemory). Muistia varatessa ensimmäisenä tulee varata laitemuistiobjekti, toisena luoda puskuri tai kuva objekti ja lopulta sitoa muisti ja puskuri tai kuva yhteen käyttäen vkBindBufferMemory tai vkBindImageMemory funktioita. Laitemuistiobjektin luonti on pakollista, koska se esittää tietyn pituista lohkoa muistia, joka on varattu tietyn tyyppisestä muistista, sellaisesta, jota fyysinen laite tukee. Yleisesti ottaen Vulkanissa on parempi varata isompi lohko muistia yhteen laitemuistiobjektiin ja asettaa useampi puskuri tai kuva siihen. Muistin varaus on jokseenkin kallis operaatio ja fyysisellä laitteella on rajoitettu määrä muistin varauskertoja.

Aikaisemmin totesimme, että meidän tulee aina varata ja sitoa laitemuistiobjekti jokaiselle kuvalle, mutta tämä ei päde swapchainin kohdalla. Swapchain edustaa lopullista kuvaa, jonka käyttöjärjestelmä lopulta piirtää näytölle. Swapchainin luonti on siis alusta kohtaista (“Vulkan® 1.3.246 - A Specification” 2023). Vulkan ei siis suoranaisesti tarjoa tapaa piirtää kuvia näytölle. Onneksi nykyään Vulkan sisältää lisäosia (“Vulkan WSI Swapchain” 2023) jotka mahdollistavat tämän kätevästi, näitä lisäosia kutsutaan yleisesti lyhenteellä WSI, Windowing System Integration. WSI-lisäosat tulee luonnollisesti ottaa käyttöön Vulkaninstanssia luodessa, meidän tulee myös ohjelmoijina selvittää, onko WSI-lisäosat tuettuja kyseisellä alustalla, jolla instanssia ollaan luomassa. Tämä on tosin helppoa, jos jo käytetään ikkunanhallintakirjastoa kuten GLFW, SFML, SDL tai SIGIL. Kun lisäosat on tarkistettu ja instanssi luotu WSI-lisäosailla, voimme luoda pintaobjektin (VkSurfaceKHR). Luonti tapahtuu tuttuun tapaan Vulkaninstanssin avulla, sekä alustariippuvaisella SurfaceCreateInfo parametrilla. Tähän CreateInfo-objektiin täytetään esim. Windowsilla instanssi (HINSTANCE) ja natiivi ikkuna (HWND) (Lapinski 2017). Voidaan ajatella pintaobjektin esittävän Vulkanissa ikkunaa. Nyt kun meillä on pinta ja laite objekti, voimme luoda Swapchain-objektin näiden avulla. Swapchain sisältää useita kuvia, esim. kaksin- tai kolminkertaista puskurointia varten. Nämä kuvat ovat jo Vulkanin varaamia, joten meidän ei tarvitse huolehtia manuaalisesta

varauksesta.

3.5 Kuvaajat ja näkymät

Puskureita ja kuvia ei aina käytetä suoraan renderöinnissä. Yleensä näiden päällä käytetään vielä erästä tasoa nimeltä näkymät (VkImageView). Näkymien perusidea on sama kuin näkymillä yleensä esimerkiksi tietokannoissa. Ne edustavat kokoelmaa parametreja, joilla voimme tarkastella dataa halutulla tavalla. Puskurinäkymä on objekti, joka luodaan tutkittavan puskurin avulla. Voimme spesifioida tietyn kohdan ja alueen, jota puskurista halutaan tutkia. Kuvanäkymät toimivat samalla periaatteella. Pikseleitä voidaan tutkia tietyssä formaatissa, komponentteja voidaan sekoitella tai näkymä voidaan rajoittaa tiettyihin MIP-tasoihin. Jotta varjostimissa voidaan käyttää puskureita ja muita resursseja, täytyy meidän käyttää kuvaajia (VkDescriptors). Kuvaajat eivät voi elää yksinään vaan ne ovat aina niputettuina kuvaajakokoelmiin (VkDescriptorSets). Kuvaajakokoelmat luodaan spesifioimalla kuvaajakokoelmarakenne (VkDescriptorSetLayout). Rakenne voi määrittää varjostimissa esim. vakioita, kuvia ja puskureita. Meidän tulee vielä luoda kuvaaja varasto (VkDescriptorPool). Varasto on yksinkertainen objekti kuvaajakokoelmien alustukseen. Varastoa luotaessa tulee spesifioida kuvaajakokoelmien enimmäismäärä ja haluttujen kuvaajien tyypit, joita ollaan varaamassa. Kuvaajakokoelmien luonti tehdään yleensä ohjelman materiaalisysteemin avulla, jotta tiedetään halutut kuvaajakokoelmat etukäteen. Lopulta voimme varata kuvaajakokoelman. Tarvitsemme siis varaston ja kuvaajakokoelmanrakenteen. Kuvaajakokoelma siis edustaa muistia, johon itse kuvaajat on varattu. Voimme konfiguroida kuvaajat osoittamaan tiettyihin puskureihin, puskurinäkymiin, kuviin tai samplereihin. Tämä tapahtuu käyttämällä "vkUpdateDescriptorSets-funktiota. Useita kuvaajakokoelmia voidaan asettaa aktiiviseksi komentopuskuriin, jotta niitä voidaan käyttää renderöintikomennoissa. Tämä onnistuu "vkCmdBindDescriptorSets-funktiolla. Tosin tarvitsemme toista objektia esittämään haluttua liukuhihnarakennetta (VkPipelineLayout). Tämän objektin avulla määrittelemme Vulkanille mitä ja kuinka montaa, kuvaajakokoelmaa ollaan käyttämässä. Liukuhihnarakenneobjekti määrittelee renderöintiliukuhihnan konfiguraation, eli minkä tyyppiset kuvaajakokoelmat ollaan sitomassa komentopuskuriin. Luomme tämän taulukosta kuvaajakokoelmarakenteita.

4 Vulkan ja OpenGL eroavaisuudet

4.1 Arkkitehtuuri

OpenGL-rajapinta käyttää niin sanottua välittömän tilan lähestymistapaa renderöinnissä. Jokainen renderöintikomento annetaan juuri siinä järjestyksessä, kun niiden halutaan tapahtuvan. Tämä ei ole mahdollista Vulkanissa, sillä moniydin tuki olisi mahdotonta implementoida. Meidän tulee sen sijaan suunnitella jokaisen kehyksen (frame) rakenne ja organisoida se vedoksiin (passes) ja osavedoksiin (subpasses). Osavedokset eivät ole erillisiä objekteja Vulkanissa, mutta ne ovat tärkeä osa renderöintiä Vulkanissa. Onneksi meidän ei tarvitse tietää kaikkia pieniä yksityiskohtia suunnitellessamme renderöintiliukuhihnaamme ("Understanding Vulkan Objects - Adam Sawicki" 2017). Tämän takia Vulkan antaa meille vapauden kerätä komentopuskuria useammalla ytimellä tai muutoin kokoamaan renderöintikomentoja eri järjestyksessä kuin ne halutaan lopuksi suorittaa. Vulkan on siis loistava valinta pelimoottoreja luodessa, sillä peleissä kulissi usein muuttuu pelin edetessä ja harvoin pysyy samana edes sekunnin ajan.

Usein pelikehityksessä halutaan tehdä jotakin erityisen spesifillä tavalla, jolloin OpenGL ei ole paras valinta, johtuen sen abstraktista arkkitehtuurista. Vulkanin käyttö on paljon läpinäkyvämpää kehittäjälle tässä suhteessa, joten se soveltuu paremmin reaaliaikaiseen kehitykseen. Vulkanin monisanaisemmasta luonteesta johtuen, rajapinta on jokseenkin monimutkainen kehittäjille, joilla on vähemmän kokemusta grafiikkaohjelmoinnista. Tästä syystä useat kokeneet kehittäjät suosittelevat OpenGL-rajapintaa vasta-alkajille. OpenGL tekee ajuritasolla paljon enemmän ohjelmoijan puolesta, mikä johtaa nopeampaan kehitykseen, mutta ennen kaikkea suoraviivaisempaan renderöintitekniikoiden omaksumiseen.

4.2 Suorituskyky

Vulkanin avulla voidaan renderöidä raskaampia maailmoja tehokkaammin kuin OpenGL-rajapinnalla (Lujan ym. 2021). Tutkimustuloksien perusteella Vulkan tarjoaa ennustettavamman suorituskyvyn, paremman energiatehokkuuden ja yleisesti suuremman tehokkuuden

kuin OpenGL (Lujan ym. 2019). Voidaan siis todeta, että Vulkanin avulla saadaan enemmän irti käytetystä laitteistosta kuin käyttämällä OpenGL-rajapintaa. Jos edes jokin näistä Vulkanin pääominaisuuksista on tärkeä projektille, niin Vulkanin tulisi olla lähes yksinomaan ainoa valinta. Ohjelmoijan tulee siis ottaa huomioon, kuinka raskaita taakkoja ollaan renderöimässä. Vulkan tarjoaa suurimmat hyödyt OpenGL-rajapintaan verrattuna isoissa renderöintitehtävissä, mutta saattaa hävitä suorituskyvyssä, kun renderöidään yksinkertaisia näkymiä (Lujan ym. 2021). Siispä OpenGL on saattaa olla parempi valinta yksinkertaisissa projekteissa, sillä se tarjoaa paremman suorituskyvyn ohella nopeamman kehitysprosessin kuin Vulkan.

4.3 Varjostimet

OpenGL käyttää OpenGL varjostin kieltä (OpenGL Shading Language, GLSL). Varjostimet käännetään ajovaiheessa, joten niiden kääntäminen ja lataaminen on hidasta ohjelman käynnistyksen ohessa. Vulkan parantaa tätä kääntämällä GLSL varjostin koodin tai monen muun varjostin koodin (“SPIR-V Support” 2021) binääriseksi välikieleksi nimeltä SPIR-V (Standard Portable Intermediate Representation). Näin varjostimet voidaan kääntää valmiiksi binääritiedostoiksi, jotka voidaan ajon alussa ladata levyltä tai ne voidaan jopa upottaa suoraan isäntäkieleen, jolloin varjostimia ei tarvitse edes lukea levyltä erillisistä tiedostoista. Vulkanissa varjostin resurssien hallinta on pitkälti kehittäjän vastuulla toisin kuin OpenGL:ssä, jossa ajuri optimoi varjostinresursseja kehittäjän puolesta.

4.4 Tuki ja dokumentaatio

Vulkan tukee useampia alustoja ja käyttöjärjestelmiä kuin OpenGL (“Vulkan Wikipedia” 2023). Monet modernit ominaisuudet ovat puutteellisia OpenGL:ssä, kuten esimerkiksi säteenseuranta (hardware raytracing) ja piirtovedokset (render passes). Tosin OpenGL voi myös hyödyntää Khronosin Vulkan säteenseurantalisäosia kohtuullisella vaivalla.

Khronos Group ylläpitää kattavaa dokumentaatiota molemmista rajapinnoista. Vulkanin spesifikaatiosivu (“Vulkan® 1.3.246 - A Specification” 2023) on hieman OpenGL sivua (“OpenGL Reference Pages” 2023) monisanaisempi ja ohjekirjamaisempi. OpenGL-rajapinnalle on oma

Khronos Groupin ylläpitämä wiki sivustonsa (“OpenGL Wiki” 2023), joka sisältää runsaasti tutoriaalimuotoista ohjeistusta, miten OpenGL-rajapintaa käytetään. Ylipäätään, OpenGL materiaalia on saatavilla enemmän, sillä se on pari vuosikymmentä Vulkania vanhempi. Yleisesti OpenGL apua on myös helpompi saada, jos sitä hakee hakukoneilla, keskustelupalstoilta tai pikaviestintäsovelluksilla. OpenGL tarjoaa myös enemmän kädestä pitävää ammattimaisesti tuotettua opetusmateriaalia. Vaikkakin OpenGL materiaalia on enemmän, ei silti Vulkan materiaalia ole mitenkään liian vähän. OpenGL on myös ollut standardi grafiikkarajapinta jo yli parikymmentä vuotta, joten OpenGL osaajia löytää helpommin.

4.5 Rajapinnan käyttö

Kuviossa 3 on esitelty yksinkertaisen OpenGL applikaation grafiikkaliukuhihnan alustus ja kuviossa 2 yksinkertaisen Vulkan applikaation grafiikkaliukuhihnan alustus. Pelkästään tästä pienestä esimerkistä huomaamme, että Vulkan on paljon monisanaisempi rajapinta ja kehittäjän tulee tarkasti määritellä objekteja alustaessa mitä ominaisuuksia ja toimintoja niillä on. Toisin kuin OpenGL:ssä, jossa rajapinta tekee paljon kehittäjän puolesta. OpenGL-rajapinta on pelkästään rajapinnan käytön perusteella korkeatasoisempaa kuin Vulkan. OpenGL tekee paljon päätöksiä kehittäjän puolesta ja rajapinnan käyttö on suurimmalta osin piilossa ajuri tasolla. Toisin kuin Vulkania käyttäessä kehittäjälle on selvää pelkästään koodia silmäilemällä, että mitä rajapinta tekee. Käyttökohteesta riippuen, tästä läpinäkyvyydestä on useasti hyötyä, kun kehitetään monimutkaisia renderöintiteknologioita.

```
void createGraphicsPipelineVulkan() {
  VkPipelineLayoutCreateInfo layoutInfo = {};
  layoutInfo.sType = VK_STRUCTURE_TYPE_PIPELINE_LAYOUT_CREATE_INFO;
  vkCreatePipelineLayout(device, &layoutInfo, nullptr,
     &pipelineLayout);
  // specify shader stages
  VkPipelineShaderStageCreateInfo shaderStages[] =
     {vertShaderStageInfo, fragShaderStageInfo};
  VkShaderModule vertShaderModule =
     createShaderModule(vertexShaderSource);
  VkShaderModule fragShaderModule =
     createShaderModule(fragmentShaderSource);
  VkPipelineShaderStageCreateInfo vertShaderStageInfo = {};
  vertShaderStageInfo.stage  = VK_SHADER_STAGE_VERTEX_BIT;
  vertShaderStageInfo.module = vertShaderModule;
  VkPipelineShaderStageCreateInfo fragShaderStageInfo = {};
  fragShaderStageInfo.stage  = VK_SHADER_STAGE_FRAGMENT_BIT;
  fragShaderStageInfo.module = fragShaderModule;

  // initialize rest of the graphics pipeline stages here...
  // ...

  VkGraphicsPipelineCreateInfo pipelineInfo = {};
  // ...
  vkCreateGraphicsPipelines(device, VK_NULL_HANDLE, 1,
                                  &pipelineInfo, nullptr,
                                     &graphicsPipeline);
  // bind the pipeline and initiate the rendering process
  vkCmdBindPipeline(commandBuffer,
     VK_PIPELINE_BIND_POINT_GRAPHICS, graphicsPipeline);
  vkCmdDraw(commandBuffer, 3, 1, 0, 0);
}
```

Kuvio 2: Vulkan grafiikkaliukuhihnan alustus

```
void createGraphicsPipelineOpenGL() {
  // create and compile the vertex shader
  GLuint vertexShader = glCreateShader(GL_VERTEX_SHADER);
  glShaderSource(vertexShader, 1, &vertexShaderSource, NULL);
  glCompileShader(vertexShader);

  // create and compile the fragment shader
  GLuint fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
  glShaderSource(fragmentShader, 1, &fragmentShaderSource, NULL);
  glCompileShader(fragmentShader);

  // create and link the shader program
  GLuint shaderProgram = glCreateProgram();
  glAttachShader(shaderProgram, vertexShader);
  glAttachShader(shaderProgram, fragmentShader);
  glLinkProgram(shaderProgram);

  // set the vertex attribute pointers
  glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 *
     sizeof(GLfloat), (GLvoid*)0);
  glEnableVertexAttribArray(0);

  // bind the shader program and activate the pipeline
  glUseProgram(shaderProgram);
  glDrawArrays(GL_TRIANGLES, 0, 3);
}
```

Kuvio 3: OpenGL grafiikkaliukuhihnan alustus

5 Yhteenveto

Löysimme, että molemmat rajapinnat ovat hyviä vaihtoehtoja alustariippumattomaan kehitykseen. Vulkan tarjoaa useasti paremman suorituskyvyn ja enemmän mahdollisuuksia optimointiin, mutta OpenGL tarjoaa enemmän käyttöönottoon ja opetteluun liittyvää apua. Vulkan on tuoreempi, modernimpi ja laajemmin tuettu rajapinta, etenkin mobiilikehityksessä. Vulkan on jokseenkin monimutkaisempi ottaa käyttöön kuin OpenGL, mutta ylläpito on lähestulkoon yhtä helppoa projektin kasvaessa suureksi. Tutkielma ottaa kantaa rajapintojen välillä enimmäkseen kaupallisen kehityksen näkökulmasta ja etenkin tutkii rajapintojen eroja uusille projekteille. Meidän täytyy silti tiedostaa, että muunlaisissa projekteissa vaatimukset saattavat erota suuresti. Esimerkiksi akateemisissa projekteissa OpenGL tarjoaa paljon nopeamman alustan testata uusia grafiikka-algoritmeja ja renderöintitekniikoita, verrattuna Vulkanin monimutkaisempaan rajapintaan. OpenGL on myös helpompi ymmärtää ja tarjoaa helpomman grafiikkaliukuhihnan alustuksen ja käytön, joten se soveltuu paremmin aloitteleville grafiikkaohjelmoijille. OpenGL ei siis ole lähitulevaisuudessa häviämässä mihinkään, vaan kehittäjät tulevat käyttämään sitä vielä useita vuosia, ellei jopa vuosikymmeniä.

Useasti vanhoja ohjelmistoja halutaan päivittää nykypäivän standardeille, mutta kustannukset voivat olla todella suuria. Vaikka grafiikkarajapinnan kokonaan vaihtaminen ei olisi kannattavaa, voivat kehittäjät silti hyödyntää Vulkania, jopa vanhoissa ohjelmistoissa kirjoittamatta koko renderöintimoottoria uudestaan. Tästä voi olla hyötyä, kun halutaan siirtyä käyttämään Vulkania kokonaan OpenGL-rajapinnan siasta, mutta kaiken renderöintikoodin uudelleen kirjoittaminen ei ole mahdollista, ainakaan välittömästi. Lisätutkimusta tulee tehdä kehittämään kirjastoja, joilla kehittäjät voivat keventää siirtymistä OpenGL-rajapinnan käytöstä Vulkanin käyttöön.

Kaiken kaikkiaan ohjelmointirajapinnan valinta riippuu kehityskohteesta. Kehittäjän tulee siis punnita, että ylittävätkö Vulkanista saadut hyödyt sen monimutkaisuuden. Vulkan on lähes ainoa oikea valinta mobiilikehitykseen, sillä se tarjoaa lähestulkoon aina paremman suorituskyvyn ja tuen, sillä Vulkan on kehitetty alusta alkaen toimimaan moitteettomasti mobiilialustoilla. Useammalle pienimuotoiselle projektille OpenGL sopii loistavasti, sen helpon käytön ja käyttöönoton takia. OpenGL tarjoaa myös hyvän suorituskyvyn yksinkertaisissa

käyttötarkoituksissa. Vulkan sen sijaan on lähestulkoon ainoa oikea valinta moderneiden pelimoottoreiden tai 3D piirto-ohjelmien toteutukseen, johtuen sen tarjoamista matalantason optimointimahdollisuuksista.

Lähteet

Akeley, Kurt. 1993. “RealityEngine Graphics”.

Ioannidis, C. ja A.-M. Boutsi. 2020. “MULTITHREADED RENDERING FOR CROSS-PLATFORM 3D VISUALIZATION BASED ON VULKAN API”. *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences* XLIV-4/W1-2020:57–62. https://doi.org/10.5194/isprs-archives-XLIV-4-W1-2020-57-2020. https://www.int-arch-photogramm-remote-sens-spatial-inf-sci.net/XLIV-4-W1-2020/57/2020/.

“Khronos Group GitHub”. 2023. https://github.com/KhronosGroup.

Lapinski, Pawel. 2017. *Vulkan Cookbook.* Packt Publishing Ltd.

Lujan, Michael, Michael Baum, Dayuan Chen ja Ziliang Zong. 2019. “Evaluating the Performance and Energy Efficiency of OpenGL and Vulkan on a Graphics Rendering Server”. https://doi.org/10.1109/iccnc.2019.8685588. https://ieeexplore.ieee.org/document/8685588.

Lujan, Michael, Michael McCrary, Blake W. Ford ja Ziliang Zong. 2021. “Vulkan vs OpenGL ES: Performance and Energy Efficiency Comparison on the big.LITTLE Architecture”. Teoksessa *2021 IEEE International Conference on Networking, Architecture and Storage (NAS),* 1–8. https://doi.org/10.1109/NAS51552.2021.9605447.

“Nvidia DLSS”. 2023. https://www.nvidia.com/fi-fi/geforce/technologies/dlss/.

“OpenGL Reference Pages”. 2023. Viitattu 2. syyskuuta 2023. https://registry.khronos.org/OpenGL-Refpages/gl4/.

“OpenGL Wiki”. 2023. Viitattu 2. syyskuuta 2023. https://www.khronos.org/opengl/wiki.

“SGI Indigo Workstations”. 1998. Viitattu 1. tammikuuta 1998. http://www.megarat.com/indigo/#apps.

“SPIR-V Support”. 2021. https://www.khronos.org/spir/.

“Transitioning from OpenGL to Vulkan”. 2016. Viitattu 2. marraskuuta 2016. https://developer.nvidia.com/transitioning-opengl-vulkan.

“Understanding Vulkan Objects - Adam Sawicki”. 2017. Viitattu 8. heinäkuuta 2017. https://gpuopen.com/learn/understanding-vulkan-objects.

“Vulkan Accouncement”. 2015. Viitattu 3. kesäkuuta 2015. https://media.steampowered.com/apps/valve/2015/Pierre-Loup_Griffais_and_John_McDonald_Vulkan.pdf.

“Vulkan Release”. 2016. https://www.khronos.org/news/press/khronos-releases-vulkan-1-0-specification.

“Vulkan Tutorial”. 2023. https://vulkan-tutorial.com/.

“Vulkan Wikipedia”. 2023. Viitattu 2. syyskuuta 2023. https://en.wikipedia.org/wiki/Vulkan.

“Vulkan WSI Swapchain”. 2023. https://registry.khronos.org/vulkan/specs/1.2-extensions/html/vkspec.html#_wsi_swapchain.

“Vulkan® 1.3.246 - A Specification”. 2023. https://registry.khronos.org/vulkan/specs/1.3-extensions/html/index.html.