3D grafikus motor fejlesztése Vulkan alapokon

Önálló labor II.

Oczot Balázs  
PRAMRX

# Bevezetés

A feladatom egy 3D grafikus motor fejlesztése volt Vulkan és OpenGL alapokon, úgy hogy a program indulása előtt el lehet dönteni, hogy melyik fajta render-engine-t szeretném használni.

Volt már korábbról egy OpenGL-es játékmotorom, de ez eléggé az olasz konyha stílusában készült, tehát a kód egy része igen spagetti volt, ezeket újraírtam. Mivel a vulkánnal előtte még nem találkoztam, így ez idegen terep volt nekem, de mindenképpen ki akartam próbálni, így ennek az implementációja biztos vagyok hogy sok helyen nem hatékony és lehetne jobban is csinálni ha több tapasztalatom lenne az API-val.

Megvalósított funkciók:

* Tetszőlegesen nagy háromszögháló modell betöltése
* Textúrák használata
* Blinn-Phong árnyalás
* Gamma korrekció
* MSAAx4 alapú élsimítás
* GPU generált mipmap-ek

Cél volt még hogy a rendszer rugalmas és könnyen használható legyen, illetve eredetileg egy minecraft szerű világotot szerettem volna elkezdeni implementálni benne, ám ez a terv túl ambíciózusnak bizonyult, és így ennek megvalósítása még várat magára.

A projekt vulkan-os részét legfőképpen a vulkan-tutorial.com oldalon található tutorial alapján készítettem el, és teljesen meg voltam elégedve vele, mert nagyon érthetően és követhetően magyaráztott mindent. Az OpengGL-es részek leginkább a learnopengl.com és a opengl-tutorial.org alapján készültek.

# Használt könyvtárak

Számos függősége van az elkészült projektnek:

* *glfw3*

Ablak létrehozáshoz és a felhasználói bemenetek és események kezelésére

* *glm*

Header-only matematikai könyvtár kifejezetten a számítógépes grafikához szükséges műveletek és struktúrák támogatására

* *vulkan-sdk*

Biztosítja a vulkan api-ban elérhető függvényeket a gpu vezérlésére

* *vulkan-hpp*

Header-only c++ nyelnyű csomogaló a c nyelvű vulkan api-hoz, ami megkönnyíti a fejlesztők elétét és még csak futásidejű költsége sincs

* *glad*

OpenGL függvény betöltő generátor

* *fmt*

Modern string formázó könytár

* *tinyobjloader*

Obj modell fájlok hatákony és gyors betöltésésre képes könyvtár

* *rapidjson*

Egy json fájl betöltő és generáló

* *stb-image*

Képek betöltésére és írására használható könyvtár

# Fejlesztőkörnyezet

A projektet windows-on fejlesztettem windows-ra, de az összes függősége cross-platform és belül sem használok semmilyen platform specifikus elemet, így valószínűleg egyből vagy kevés munkával még linuxon is lefordulna.

A buildsystem cmake alapú, és legalább 3.17-es cmake kell, mivel használom a precompiled header funkcióját a cmake-nek.

A függőségek legkönnyebben a vcpkg program használatával telepíthetők fel, most éppen a ”2020.11-1”-es kiadást használom, ha minden függőség innen van telepítve, akkor ezzel biztosan fordul a programom.

Mivel a Visual Studio 2019 már natívan támogatja a CMakeList.txt betöltését, így elég csak megnyitni a projekt mappáját, és már lehet is fordítani.

Alapvetően C++20 standarddal bekapcsolva fejlesztettem, de lehet elég neki a C++17 is hogy leforduljon, de a 16.8.3-mas verziójú Visual Studio-val ennek is biztosan működnie kell.

# Vulkan API bevezető

A Vulkan API-t 2016-ban adta ki a Khronos Group, azóta már az 1.2-es kiadás is elérhető, és napjainkban is aktívan dolgoznak a projekt fejlesztésén.

Az OpenGL-t tekinthető a projekt elődjének, de nem mondanám azt, hogy egy-az-egyben leváltani szeretnék, csak inkább egy külön utat biztosítanak a gpu-k hatékony kezeléséhez, de ez kompromiszumokkal jár.

Mivel az OpenGL egy elég régi szabvány, így hiába javítgatják az új kiadásokkal, az alap koncepciókat nem tudják már megváltoztatni, és emiatt nem alkalmas a mai gpu-k vezérlésére, mert azok architektúrája jelentősen megváltozott az évek folyamán.

A Vulkan erre a problémára ad megoldást, mely egy low driver overhead mentalitással próbálja elvégezni a feladatot, így minnél több erőforrást meghagyva más folyamatoknak. Ám ez nincs ingyen, a kód sokkal hosszabb és bonyulultabb lesz, mert mindent explicit kell eldönteni hogy mit és hogyan szeretnénk használni. Erre azért van szükség, hogy ne a driver-nek kelljen találgatnia arról hogy mit is akarhat a programozó, hanem amiatt, hogy minden le van írva így már hatékonyabban és egyszerűbben lehet a driver-ben optimalizálni.

A vulkan előnye még hogy az API cross platform és egy egyesített interfészt tudnak használni a desktop és a mobil programok is.

Hogy valaki vulkan kódot tudjon futtatni szükség van szoftveres driver támogatásra és a hardveres támogatásra is. Az AMD és NVIDIA oldalon általánosan elmondható, hogy a 2012 után kiadott eszközök már támogatják, az Intelnél ez 2015, míg az Androidos eszközöknél körülbelül 2016.

# Különbségek az OpenGL-hez képest

Az OpenGL egy globális állapoton alapszik, emiatt kell mindig először ”bind”-olni az erőforrást és csak utána lehet vele dolgozni, ám ezzel szemben a Vulkan objektumorientált. A Vulkánban struktúrákat kell felkonfigurálni a kívánt adatokkal, így ezek eléggé összetettek és nagyméretűek lehetnek, úgy hogy közben még hivatkozhatják is egymást, de ennek a megközelítésnek az az előnye, hogy nem kell több külön-külön költséges függvényhívást intézni a hardver felé, hanem egyetlen függvényhívással el lehet intézni azt. Hátrány hogy ezt a rengeteg paramétert fel kell konfigurálni és nincsekek alapértelmezett értéket, tehát ez egy időigényes és komplex feladat.

A vulkanban lehetőség van opcionális validációs layer-ek használatára, melyek callback függvényhívással a programozó tudtára adják hogyha valamit elrontott, de még akár figyelmeztetéseket is tud küldeni, hogy máshogy kellene használni az erőforrásokat mert így nem hatékony. Nagy előnye hogy ez kikapcsolható, és míg például debug üzemmódban érdemes bekapcsolva hagyni, addig release módban nem akarunk a validációra fölöslegesen erőforrásokat pazarolni, hiszen valószínűleg úgysem tudjuk runtime lekezelni a hibát, és a vége úgyis az lesz, hogy összeomlik a program. Az OpenGL esetében a valódációt mindig a driver végezte.

# Legfontosabb erőforrások

* Instance: Összektöti az alkalmazást és a vulkan konyvtárat, és leírja hogy milyen bővítményeket használnánk
* Phisical device: Fizikai gpu kiválasztása, le lehet kérdezni a kártya adottságait, pl ram méret, támogatott funkciók
* Logical device: A phisical device-al tartja a kapcsolatot, pontosan leírja hogy milyen gpu funkciókat akarunk használni
* Queue, Queue family: Az egyes queue-k különböző parancsokat támogatnak, így a parancsoknak megfelelő queue-t kell választani, mivel ezekből több is van amik másra használhatóak (pl grafikus megjelenítés, compute parancsok, memory transfer parancsok, stb…)
* Surface: Ablak a megjelenítéshez
* Swap chain: A folyamatos megjelenítés miatt egyszerre több képen kell dolgozni. Páldául ha egy renderelt frame elkészült akkor azt meg kell mutatni a felhasználónak minnel gyorsabban, de közben a következő frameüen már el kell kezdeni dolgozni. Ezeket biztoítja a swapchain. Double buffering, Triple buffering támogatásra való.
* Rendess pass: Leírja az egyes attachment-ek (pl colorbuffer, framebuffer) típusát, használatát, és hogy hogyan kell ezeket kezelni egy renderelési lépésnél
* Subpass: A render pass lépésekre bontása, azért jók mert elérhetik a korábbi subpass-ok eredményeit.
* Graphics pipeline: Leírja a gpu konfigurálható részeinek a működésést (pl viewport méret, depth buffer kezelés, color blending, shaderek használata)
* Command buffer: Az egyes elvégzendő műveleteket amik a queue-kba kerülnek rögzíteni kell egy command bufferba

# Háromszög rajzolás lépései

1. VkInstance elkészítése
2. Gpu kiválasztása (VkPhysicalDevice)
3. VkDevice és VkQueue elkészítése a rajzoláshoz és megjelenítéshez
4. Ablak, surface, és swapchain elkészítése
5. Swapchain képekhez VkImageView referencia készítése
6. Renderpass elkészítése
7. Framebufferel elkészítésére a renderpass-hoz
8. Graphics pipeline elkészítése
9. A rajoló parancsok allokálása és felvétele minden swapchain képhez
10. Üres kép kérése a swapchain-től, majd command buffer beküldése, és a kész kép visszarakása a swapchain-be

# Szinkronizáció

# A queue-kba beküldött parancsok aszinkron futnak, így szinkronizációra is szükség van a helyes végrehajtáshoz, például hogy ne kezdjünk el egy műveletet addig, amig az ahhoz szükséges input nincs készen.

# A Vulkan-ban kétfajta szinkronizációs primitív található:

* Fence
  + CPU-GPU között
  + Explicit lehet rá várni vkWaitForFence függvénnyel
* Semaphore
  + GPU-GPU között
  + Nem lehet rá várakozni explicit a programból, de a command queue-kban használhatóak a parancsok szinkronizálására egyfajta előkövetelményként

# Memória kezelés

A Vulkanban a memóriakezelés teljesen explicit módon működik, tehát mindent a programozónak kell megcsinálnia, ezzel szemben OpenGL esetében ezt a munkát a driver végzi. Azonban a ”Semmi gond, tudom mit csinálok” megközelítésnek hála hatékonyabban lehet a memóriát kihasználni számos esetben.

Egyik lehetőség, hogy általában jobban megéri újra felhasználni egy objektumot, mint újra létrehozni azt.

A vulkanban különböző memóriatípusok különböző erőforrásokkal kompatibilisek, vagy csak simán előnyösebb a használatuk, így erre is figyelni kell.

Bevett szokás a staging buffer használata is, azaz amikor a cél buffer nem képezhető le egy az egyben a cpu címterébe (például nem egyezik a formátum), akkor egy köztes buffert kell csinálni és oda kell felmásolni az adatokat, majd pedig ezután a köztes bufferből konvertálással lehet a végleges helyére juttatni az adatot.

A cache lokalitás miatt nem érdemes mindennek külön-külön apró buffereket csinálni, hanem inkább érdemes egy darab nagyobbat lefoglalni, és abban okosan elhelyezni az egyes elemeket.

Lehetőség van memory alising használatára is, tehát hogy egy időben több resource is ugyanazt a területet használhatja, de ez nem feltétlenül gond, mert valójában csak az egyikük aktív bármely időpillanatban.

# Több render engine backend

Csináltam egy alap c++ programot, ami két render engine backend-et is használni tud. Az egyik a Vulkan, a másik pedig OpenGL. Mivel nincs a programomba játék logia, így a kód nagy részéért a backendek felelősek, tehát asset betöltő részt és vagy 10 közös interface függvényt leszámítva két külön program fut alatta. Ezt nem tartom rossz megoldásnak, hiszen bármilyen másik architektúrát használva kevésbé lenne flekszibilis a rendszer, és ez gyakran jól jött.

# Modell megjelenítés benchmark

A fejlesztés során nem tartottam kifejezetten fontosnak, hogy minden a végletekig ki legyen optimalizálva, de azért még így is kíváncsi voltam, hogy a két render engine mit tud, ha összehasonlítom egymással a sebességüket.

Több modellt is megnéztem, az egyik nagyobb az a Stanford-féle sárkány volt, aminek 871306 háromszöge van. Mindig 2560x1080 felbontáson futtattam a programot, és egy gtx 1080 Ti-t használtam. Számomra először kicsit meglepően, de mindkét esetben 2150 fps-en futottak, de más modellek esetén is kb fej-fej mellett voltak. Azonban én ezt annak tudom be, hogy nem volt nagy driver overhead OpenGL oldalon sem, mivel egy egyszerű megjelenítést használtam, és ha nem így lett volna akkor jobban megmutatkozott volna a Vulkan előnye.

# Vulkan vagy OpenGL-t használjak?

Ki használjon Vulkan-t?

* Aki minden teljesítmény ki akar sajtolni a gpu-ból, például egy nagy AAA játékfejlesztő stúdió
* Ha valakit a grafika érdekel, nem pedig a játékfejlesztés rész

Ki ne használjon Vulkan-t?

* Nincs sok (rengeteg) ideje
* Komplex lesz a kód, végtetekig optimalizált esetben hibalehetőségekkel tele
* Nehezen debugolható szinkronizáció
* Könnyű benézni dolgokat, és apróságok miatt akár rosszabb teljesítmény is kapható mintha valaki csak simán OpenGL-t használna

# Továbbfejlesztési lehetőségek

Eredetileg egy minecraft-szerű játékot szerettem volna elkezdeni fejleszteni, de a vulkan api mélyebb és időigényesebb volt, mint ahogy gondoltam, így erre még nem jutott idő. Azonban a közeljövőben ha arra kerülne a sor, akkor implementálni lehetne kezdésnek egy 16x16-os chunk-on alapuló procedurálisan generált világot, úgy tudom hogy eredetileg is ezzel a chuck-os technikával készült a játék és a mai napig is ezt használják.

Akár egy skeletal alapó karakter animáció is hasznos lenne, ezáltal a világ kicsit élettel telibb lenne, mivel így nem csak statikus objektumok lennének benne, hanem akár állatok is ugrándozhatnának a szabadban.

Jó minúségű árnyékok megjelenítése szintén hasznos lenne.

# Referenciák

1. <https://vulkan-tutorial.com/>
2. <https://github.com/KhronosGroup/Vulkan-Hpp>
3. <https://github.com/tinyobjloader/tinyobjloader>
4. <https://developer.nvidia.com/vulkan-memory-management>
5. <https://learnopengl.com/>
6. <http://www.opengl-tutorial.org/>