Szegedi Tudományegyetem
Informatikai Intézet

SZAKDOLGOZAT

Horváth Levente

2025

Szegedi Tudományegyetem
Informatikai Intézet

Vulkan renderelő motor

Szakdolgozat

Készítette:
Horváth Levente
informatika szakos hallgató

Témavezető:
Dr. Kiss Ákos
egyetemi docens

Szeged
2025

Vulkan renderelő motor

FELADATKIÍRÁS

A hallgató feladata egy új generációs grafikus API (Vulkan, Metal, stb.) felhasználásával egy úgynevezett renderelő motor megvalósítása, amely megfelelően elkésztett grafikus csővezeték segítségével képes komplex, textúrázott 3D objektumok megjelenítésére és mozgatására, valamint különböző fényforrások szimulálására.

Vulkan renderelő motor

TARTALMI ÖSSZEFOGLALÓ

- **Célok:**

  A szakdolgozat célja egy Vulkan-alapú grafikus motor fejlesztése, amely képes külső 3D modellek betöltésére, megjelenítésére és alapvető fénykezelési technikák alkalmazására.A Vulkan egy alacsony szintű, nagy teljesítményű grafikus API, amely teljes kontrollt biztosít a GPU erőforrásai felett, így hatékonyabb renderelési megoldásokat tesz lehetővé, ugyanakkor összetettebb fejlesztési folyamatot igényel az elődeihez képest. A projekt során elsődleges célként a Phong-féle világítási modell implementálása szerepelt, amely három fő világítási komponenst (ambient, diffuse és specular) foglal magában.

- **Eszközök:**

  A fejlesztés során a GLFW könyvtárat használtam az ablakkezeléshez és az Open Asset Import Library (Assimp) segítségével külső 3D modellek betöltését valósítottam meg. A használt nyel a C++. A fejlesztés során, a Visual Studio 2022 integrált fejlesztő környezetet használtam. A projekt végére sikerült egy működőképes Vulkan-alapú renderelő motort létrehozni, amely támogatja a több objektum egyidejű betöltését és megjelenítését, valamint a Phong-féle világítási modell használatát.

- **Eredmények:**

  A projekt sikeresen be tud olvasni, meg tud jeleníteni és tud mozgatni komplex objektumokat. Képes az objektumokat textúrázni és Phong világítást kezelni.

- **Kulcsszavak:**

  Vulkan, grafikus motor, modellek betöltése, shader programozás, fénykezelés, textúrázás.

TARTALOMJEGYZÉK

Feladatkiírás *1*

Tartalmi összefoglaló *2*

Tartalomjegyzék *3*

1. Bevezetés *4*

1.1. Vulkan 4

1.2. Miért Vulkan? 5

2. Felkészülés *6*

2.1. Ablak 6

2.2. A képkocka-puffer szerepe 6

2.3. Kamera 7

3. Vulkan felállítása *8*

3.1. Példány (Vulkan Instance) 8

3.2. Utasítási sorok (Queues) és sorcsaládok (Queue Families) 8

3.3. Vulkan képmegjelenítés 9

3.3.1. Cserelánc (Swapchain) és felszín (Surface) 9

4. Grafikus csővezeték (graphics pipeline) *12*

4.1. Grafikus csővezeték létrehozása 13

4.2. Renderelési fázis (Render Pass) és alfázisai (Subpasses) 14

4.3. GPU parancsok végrehajtása 14

4.4. Szinkronizálás, zászlók (Flage) és Szemaforok (Semaphore) 15

4.5. Vertex adatok shaderhez küldése 16

4.6. Adatküldés a shadereknek 17

4.7. Depth Buffer 18

4.8. Textúrázás 19

5. Külső objektumok beolvasása *21*

5.1. Hálókezelő osztály 21

5.2. Adat beolvasás 22

6. Fények kezelése és a Phong-féle világítási modell *24*

7. Összegzés *27*

Irodalomjegyzék *28*

Nyilatkozat *29*

Köszönet nyilvánítás *30*

Mellékletek *31*

BEVEZETÉS

A számítógépes grafika az elmúlt évtizedekben hatalmas fejlődésen ment keresztül, amelynek során a korábban forradalminak számító technológiák, mint a Doom vagy a Build Engine, mára megszokottá, sőt elavultá váltak.

Úgy éreztem, ennek ellenére az egyetemi oktatásban a grafikus renderelés elméleti és gyakorlati háttere sokszor nem kap kellő figyelmet annak ellenére, hogy ez a téma rendkívül komplex és a mindennapjainkban is megkerülhetetlen. A szakdolgozatom célja egy saját grafikus renderelő motor fejlesztése, amely képes komplex 3D objektumok megjelenítésére és mozgatására, valamint különböző fényforrások szimulálására. A fejlesztési folyamat során a hatékony és optimalizált működés elsődleges szempont volt, hiszen a modern grafikai API-k egyik fő előnye az erőforrások megfelelő kihasználása.

Manapság rengeteg hatékony és elterjedt grafikus API-ok vannak, mindegyik a saját előnyeivel és hátrányaival. Ilyenek például a Direct3D 12, Metal, vagy az OpenGL.

A projekt megvalósításához a Vulkan API-t választottam, amely alacsony szintű, erőforráshatékony és többplatformos grafikus API, kifejezetten 3D megjelenítéshez optimalizálva. Emellett számos külső könyvtárat is felhasználtam a fejlesztés során.

A GLFW könyvtár az ablakkezelés és a grafikus kontextus inicializálásának megkönnyítésére szolgált, a GLM a matematikai számításokat támogatta, míg az Assimp lehetővé tette a 3D modellek importálását. A renderelő motor C++ nyelven készült, mivel ez a nyelv biztosította a megfelelő teljesítményt és rugalmasságot.

A projekt során az objektumorientált programozás és a grafikus programozás alapelveit követve építettem fel a rendszert, amely képes valós idejű 3D környezetek kezelésére, textúrázott objektumok megjelenítésére.

A fejlesztési folyamat során szerzett tapasztalatok hozzájárultak a modern grafikus API-ok mélyebb megértéséhez, valamint a hatékony renderelési stratégiák kialakításához. Az elkészült rendszer demonstrálja a Vulkan API által nyújtott lehetőségeket, miközben bemutatja a grafikai feldolgozás kulcsfontosságú lépéseit és az optimalizált adatkezelés előnyeit.

1.1. Vulkan

Mielőtt nekikezdhetnénk a munkálatoknak, meg kell értenünk, hogy mi is a Vulkan. A Vulkan önmagában egy API-specifikáció, amelyet a Khronos Group fejleszt és tart karban. Ez

azt jelenti, hogy a Vulkan maga nem egy konkrét implementáció, hanem egy szabvány, amely meghatározza, hogyan kell egy grafikus API-nak működnie. Az implementációt a GPU-gyártók valósítják meg saját illesztőprogramjaik formájában. Egy Vulkan program tehát a használt GPU gyártójának Vulkan-implementációjára támaszkodik.

A Vulkan futtatásához szükség van egy Vulkan-kompatibilis GPU-ra, a megfelelő illesztőprogramra és a Vulkan SDK-ra. A projekthez én a LunarG Vulkan SDK 1.3.290.0 verzióját használtam. Ez az SDK tartalmazza a fejlesztéshez szükséges elemeket, például a

Vulkan fejléceket, könyvtárakat, validációs rétegeket, amelyek a hibakereséshez és a fejlesztés közbeni ellenőrzéshez szükségesek, a SPIR-V eszközöket, amelyek a shader fordításához és ellenőrzéséhez szükségesek és a Vulkan dokumentációt és példaprogramokat, illetve számos további eszközt, amelyek a fejlesztés különböző fázisait segítik elő.

1.2. Miért Vulkan?

Számos kiváló grafikus API létezik, mindegyiknek megvannak a maga előnyei és hátrányai. A projektem elkészítéséhez a Vulkan-t választottam, mert ennek a grafikus és számítási API-nak az előnyei tűntek a legmegfelelőbbnek az általam kitűzött célok eléréséhez. Mivel egy olyan programot szerettem volna készíteni, amelyet később tovább tudok fejleszteni, fontosnak tartottam, hogy egy cross-platform API-t válasszak, amely több operációs rendszeren és hardverplatformon is működik.

Az elterjedt grafikus API-k közül csak az OpenGL és a Vulkan biztosít széles körű támogatást több platformon. A Metal sajnos csak Apple-eszközökön érhető el, míg a Direct3D kizárólag Windows és Xbox platformokon használható. Így a két lehetséges választási lehetőségem a Vulkan és az OpenGL volt.

Az OpenGL azonban egy elavulóban lévő technológia, amely korlátozott eszközöket biztosít a GPU-menedzsmenthez. Ezzel szemben a Vulkan egy modernebb API, amely nagyobb szabadságot ad a fejlesztők kezébe a GPU kezelésében. Ennek azonban az a következménye, hogy mivel a GPU-k működése meglehetősen komplex, a fejlesztőnek kell közvetlenebb módon kezelnie ezt a komplexitást, ami több kódsor megírását teszi szükségessé.

2. FELKÉSZÜLÉS

2.1. Ablak

Mielőtt azonban elkezdhettem volna dolgozni a tényleges grafikus motoron, először szükségem volt egy ablakra, amelyre elkezdhettem rajzolni. Azonban, mivel az ablak létrehozása és kezelése operációs rendszertől függő feladat, a Vulkan szándékosan el van különítve ettől, hogy megőrizze a platformfüggetlenséget. Emiatt szükségem volt egy másik eszközre, amely biztosítja az ablakkezelést. Erre a célra a GLFW könyvtárat választottam.

A GLFW (Graphics Library Framework) egy segédkönyvtár, amelyet elsősorban OpenGL-lel való használatra terveztek. A GLFW egyszerű módot biztosít egy ablak létrehozására, annak kezelésére, valamint a bemeneti eszközök, mint például az egér és billentyűzet, kezelésére.

Az objektumorientáltság érdekében az ablak létrehozásához és kezeléséhez szükséges adatokat és függvényeket egy Window osztályba szerveztem, amely a GLFW által biztosított eszközöket használja.

Az ablak létrehozását meg kell előznie a GLFW inicializálásának, hogy a GLFW függvényei használhatók legyenek. Ezen kívül jelezni kellett a GLFW számára, hogy nem OpenGL-t fogok használni, mivel a GLFW alapvetően az OpenGL-hez lett kifejlesztve.

Ezután szükség volt az ablak nevének, szélességének és magasságának meghatározására. Ezekkel az adatokkal létre tudtam hozni egy ablakot. Azonban ez önmagában még nem elég. Ha egy ilyen egyszerűen létrehozott ablakra próbálnánk meg kirajzolni egy képet, gyorsan észrevennénk, hogy a kép nem egyszerre jelenik meg, hanem a bal felső sarokból indulva fokozatosan rajzolódik ki.

2.2. A képkocka-puffer szerepe

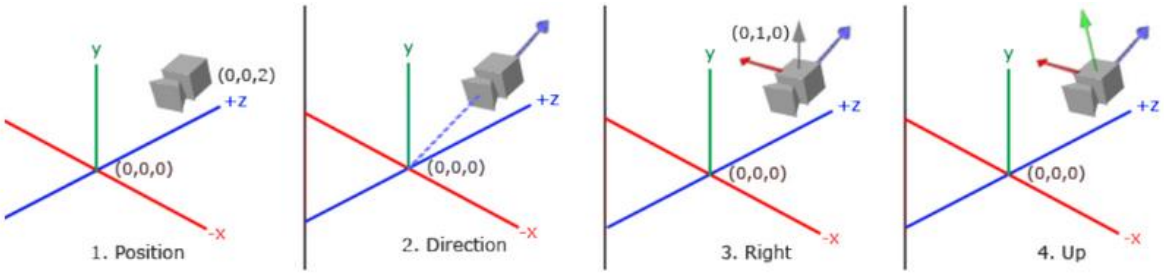
Ezt a szaggatott megjelenítést (tearing) elkerülhetjük a GLFW saját képkocka-puffer (framebuffer) használatával.

A kép először a képkocka-pufferbe kerül, nem közvetlenül az ablakra. Amikor a teljes kép elkészült, a rendszer a puffer tartalmát átmásolja az ablakba. Ez a pufferezés segít abban, hogy a kép folyamatosan és akadásmentesen jelenjen meg. Fontos figyelni arra, hogy az ablak mérete és a képkocka-puffer mérete kompatibilisek legyenek, különben grafikai hibák jelentkezhetnek. Miután az ablakom elkészült, elkezdtem dolgozni a kamerarendszeren.

2.3. Kamera

A kamera kezelésére egy egyszerű osztályt készítettem, amely lehetővé teszi a kamera mozgatását és irányítását. Amikor a kameráról vagy a nézeti térről beszélünk, minden objektum helyzetét a kamera szemszögéből nézzük. Ehhez volt szükség egy nézeti mátrixra. A nézeti mátrix átalakítja az összes objektum helyzetét úgy, hogy azok a kamera pozíciójához és nézeti irányához képest legyenek meghatározva.

A kamera létrehozásához és működéséhez több dologra volt szükség. Először szükségem volt a kamera világ térbeli koordinátájára, hogy tudhassam hol van egyáltalán a kamera. Emellett szükségem volt egy vektorra, ami a kamerától fölfele és egy vektorra, ami a kamerától jobbra mutat. Ezek mellett persze egy irány érték is kellett, hogy tudni lehessen merre néz a kamera.



ábra 2.1
Kamera definiálás

Ha jobban belegondolunk, ezzel egy saját koordináta-rendszert hozunk létre, amelyben a kamera lesz a kiindulópont, és három egymásra merőleges irány határozza meg, hogy mi merre van a kamera nézőpontjából.

Ezen felül, mivel a kamerát mozgathatóvá akartam tenni, meg kellett határoznom egy forgási és egy mozgási sebességet. A kamera térbeli mozgását a billentyűzet segítségével lehet irányítani, míg a kamera forgatását az egér vezérli.

A kamera számára szükséges bemeneti adatokat az ablak objektumon keresztül kapja meg. Most pedig, hogy volt egy kamerám és egy ablakom el tudtam kezdeni a Vulkan renderelő elkészítését.

3. VULKAN FELÁLLÍTÁSA

3.1. Példány (Vulkan Instance)

A renderelő létrehozásához először létre kellett hozni a Vulkan példányt. Ez egy olyan objektum, amely kapcsolatot hoz létre az alkalmazás és a Vulkan környezet között. Ennek segítségével az alkalmazás hozzáférhet a rendelkezésre álló fizikai eszközökhöz, például a GPU-hoz és az engedélyezett validációs rétegek információihoz. A Vulkan példány létrehozása elengedhetetlen, mivel az API hívások ezen keresztül történnek. A Vulkan kétféle eszközt különböztet meg. Fizikai eszközöket és logikai eszközöket.

A fizikai eszközök a GPU-k, amelyek tartalmazzák a használható memóriát és az utasítási sorokat. Ha a GPU memóriájához vagy egyéb hardveres erőforrásaihoz szeretnénk hozzáférni, azt mindig a fizikai eszközön keresztül kell megtennünk.

Azonban a fizikai eszközökkel nem lehet közvetlenül interakcióba lépni, ehhez szükség van egy logikai eszközre, amely egy interfész a kommunikációhoz.

Miután sikeresen létrehoztam a Vulkan példányt, lekérdeztem a rendelkezésre álló eszközök listáját, és kiválasztottam az első megfelelő GPU-t. Ez lett az én fizikai eszközöm.

Ez azonban még nem volt elegendő. Ahhoz, hogy hatékonyan tudjak dolgozni a fizikai eszközzel, szükség volt egy további absztrakciós rétegre. Ez az absztrakciós réteg a Logikai eszköz.

A logikai eszköz egy interfész a kiválasztott fizikai eszközhöz. Ez teszi lehetővé, hogy az alkalmazás kommunikáljon a GPU-val, és utasításokat küldjön az utasítási sorokba végrehajtásra.

Minden, ami a GPU-val történik, ezen a logikai eszközön keresztül hajtódig végbe. Ez megakadályozza, hogy közvetlenül az illesztőprogramokkal kelljen foglalkoznom. Így már képes voltam utasításokat adni a GPU-nak. Ez a kódban annyit jelentett, hogy az esetek túlnyomó részében, amikor valamilyen erőforrással kívántam dolgozni, erre kellett referálnom.

3.2. Utasítási sorok (Queues) és sorcsaládok (Queue Families)

Az utasítási sorok a feldolgozásra váró parancsokat kezelik. Minden Vulkan parancs egy adott sorba kerül beküldésre, majd végrehajtásra.

Ezek a sorok FIFO (First In, First Out) módon működnek, vagyis a parancsok végrehajtása a sorba kerülésüknek sorrendjében történik

A Vulkan különböző típusú utasítási sorokat tud kezelni, amelyek különböző feladatokat látnak el. Ezeket a sorcsaládok (Queue Families).

A legfontosabb sorcsaládok a Grafikus sor (Graphics Queue) a Renderelési és rajzolási műveletek végrehajtására, a számítási sor (Compute Queue) az általános számítási feladatok végrehajtására és az átviteli sor (Transfer Queue) adatátvitelre, például textúrák vagy modellek átmásolásakor lehet szükség műveletek számára. Ezen felül léteznek ezek kombinációi is.

A projektem során egy grafikus sorcsaládot használtam, mivel az alkalmazásom főként renderelési műveleteket végez.

3.3. Vulkan képmegjelenítés

Ahogy korábban már említettem, a képek ablakra való megjelenítése nem egy beépített Vulkan funkció, mivel a Vulkan célja a többplatformosság fenntartása. Ennek ellenére a képek kezelése kulcsfontosságú egy renderelő motor működésében.

Ahhoz, hogy a renderelt kép folyamatos és szép legyen, állandóan új képkockákat kell kirajzolni az ablakra. Azonban a kirajzolás nem történik meg azonnal. Ha egy kép részleteiben jelenne meg, az zavaró és nagyon csúnya lenne. Ezért biztosítanunk kell, hogy a képernyőn mindig csak a már teljesen elkészült képek jelenjenek meg. Ezt a feladatot a kép pufferek (framebuffer-ek) látják el.

A kép elkészítésekor nem közvetlenül az ablakra rajzolunk, hanem egy puffer memóriahelyre. Így a kép csak akkor kerül megjelenítésre, amikor teljesen elkészült, és nem előbb. A teljesítmény optimalizálása érdekében több puffert is használhatunk, én három pufferes megoldást alkalmaztam. Így az én programomban egyszerre volt a Vulkan-nak kép pufferei, ahonnan a GLFW pufferébe, majd a képernyőre kerültek a képek.

Ahhoz, hogy ez így működni tudjon Vulkanban, két alapvető dologra van szükség. A csereláncra és a felszínre.

3.3.1. Cserelánc (Swapchain) és felszín (Surface)

A felszín egy interfész a GLFW által létrehozott ablak és a Vulkan csereláncban lévő kép között, biztosítva azok elkülönítését. A felszínt kifejezetten az adott ablakhoz kell létrehozni, amelyhez szerencsére a GLFW biztosít egy beépített függvényt, glfwCreateWindowSurface néven.

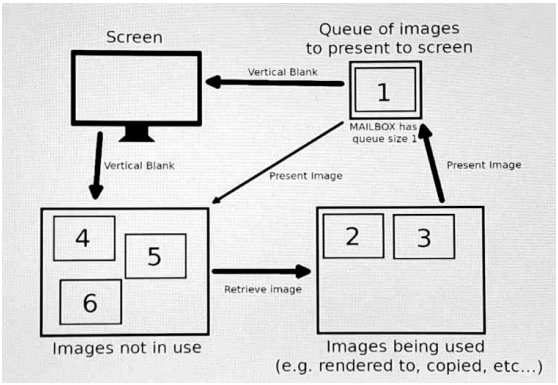
Ez a függvény segít a felszín és az ablak közötti kapcsolat létrehozásában, így a Vulkan pontosan tudja, hogy milyen ablakba kell a képeket megjeleníteni.

A cserelánc egy olyan Vulkan objektum, amely tulajdonképpen egy sorozatnyi kép, amelyekre rajzolni lehet, majd ezeket prezentálni lehet a kijelző számára. A cserelánc három fő elemből áll.

Az első a felszín képességei, amely meghatározza, hogy a cserelánc milyen típusú felülethez kapcsolódik, és milyen méretű és formátumú képeket kell kezelnie. Például itt dől el, hogy mekkorának kell lennie a megjelenítendő képeknek, hogy pontosan illeszkedjenek az ablakhoz.

A második elem a fogadó felület formátuma, amely meghatározza, hogy a képek milyen színformátumban kerülnek feldolgozásra. Például a formátum lehet RGB_NONLINEAR, mint az én programom esetén, attól függően, hogy milyen módon tárolják a színinformációkat. Ezeket az értékeket a felszín adatai alapján kell beállítani, mivel a Vulkan és a kijelző formátumainak kompatibilisnek kell lenniük, különben a kép helytelenül jelenhet meg.

A harmadik fontos elem a prezentációs mód, amely meghatározza, hogy a cserelláncban lévő képek milyen sorrendben és milyen időközönként kerüljenek ki a felszínre. A Vulkan négy prezentációs módot biztosít, amelyek közül én a VK_PRESENT_MODE_MAILBOX_KHR módot választottam.



3.1 ábra
Mailbox prezentációs mód

Ez a prezentációs mód (Present Mode) háromszoros pufferelést használ a megjelenítés optimalizálására. Úgy működik, hogy mindig a legújabban renderelt képkockát jeleníti meg, miközben eldobja az előző, még meg nem jelenített képkockákat. Ez azért jó mert ez a mód megakadájozza a képszakadást és csökkenti a késleltetést az által, hogy a korábban renderelt, de még meg nem jelenített képkockák nem várakoznak a megjelenítésre.

A VK_PRESENT_MODE_MAILBOX_KHR ezért ideális választás olyan alkalmazásokhoz, ahol fontos a folyamatos és gyors megjelenítés, például játékokhoz vagy

interaktív grafikus alkalmazásokhoz a VK_PRESENT_MODE_IMMEDIATE_KHR móddal ellentétben, ami folyamatos képszakadást tud okozni.

Most, hogy a Vulkan alapvető elemeit sikeresen felállítottam, elkezdtem a grafikus csővezetéket elkészíteni.

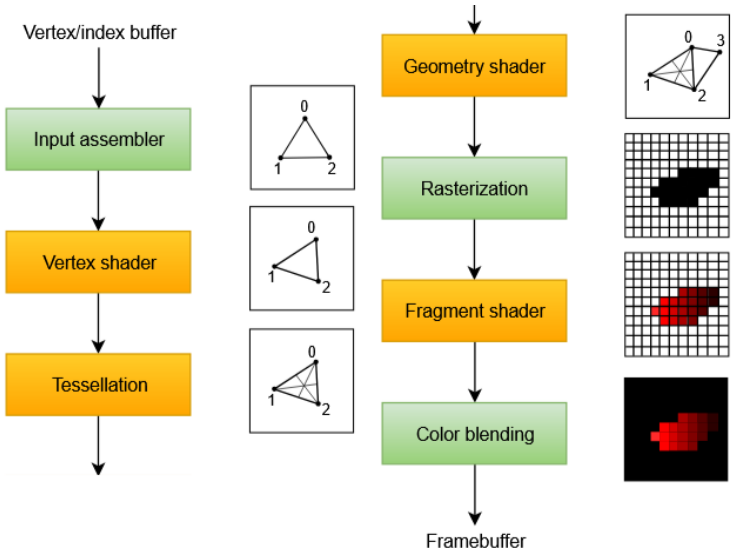
4. GRAFIKUS CSŐVEZETÉK (GRAPHICS PIPELINE)

A Vulkanban minden háromdimenziós térben kerül értelmezésre, azonban a képernyőnk csak kétdimenziós pixelekből áll. Ezért meg kell oldanunk, hogy hogyan lehet egy 3D-s teret és objektumokat ábrázolni egy 2D-s képernyőn. Ezt a problémát kezeli a grafikus csővezeték, amely a 3D adatokból képes, megjeleníthető képet létrehozni.

A grafikus csővezeték feladata, hogy a kapott háromdimenziós koordinátákat átalakítsa színes, kétdimenziós pixelekké. Ahogy a neve is sugallja, a csővezeték egy lineáris feldolgozási láncot ír le, ahol minden egyes lépés kimenete a következő lépés bemenete. Minden lépés a GPU-n futó, speciális feladatokra optimalizált programokat használ.

A shaderekről fontos tudni, hogy egy részük konfigurálhatók, míg vannak olyan shaderek, amelyeket kötelezően be kell állítani a működéshez.

A Vulkan grafikus csővezetékének főbb részei a, Bemeneti összeállító (Input Assembly), Csúcsárnyaló program (Vertex Shader), Tesszelláció (Tessellation), Geometriaárnyaló (Geometry Shader), Raszterizáció (Rasterization), Fragmentárnyaló (Fragment Shader) és Színkeverés (Color Blending). A projektem során saját Vertex Shader és Fragment Shader programokat készítettem.



ábra 4.1
Vulkan Grafikus csővezeték

A Vertex shader felelős a csúcspontok átalakításáért világtérből (world space) nézeti térbe (view space), majd eszközkoordinátákba (clip space).

A Fragment shader feladata, hogy meghatározza minden egyes pixel színét, valamint eldöntse, hogy egy adott pixel látható-e vagy sem. Itt lehet különféle effekteket is hozzáadni, például árnyékolást, tükröződést vagy áttetszőséget.

Fontos megjegyezni, hogy ezek a feladatok nincsenek kőbe vésve. Ha szeretnénk, matematikai problémák megoldására is használhatjuk a csővezetékünket, saját magunk által készített shaderek segítségével.

Shader programok és SPIR-V formátum

A shadereimet nem közvetlenül a C++ kódban írtam meg, hanem külön text formátumban készítettem el, majd SPIR-V formátumba előkompájloltam.

Ez azért szükséges, mert a Vulkan nem GLSL shadereket, hanem az előfordított SPIR-V formátumot használja. Az előkompiláláshoz szükséges eszközöket a Vulkan SDK biztosította.

Miután a shadereket SPIR-V fájlokká alakítottam, azokat bináris formátumban be kellett töltenem a C++ programba, majd a csővezeték létrehozásakor át kellett adnom ezeket az adatokat.

4.1. Grafikus csővezeték létrehozása

Egy grafikus csővezeték elkészítése, ahogyan szinte minden más a Vulkanban, egy meglehetősen sok kódot igénylő művelet.

A csővezeték konfigurálásához minden egyes fázist manuálisan kell beállítani. Például, az Input Assembly szakasz konfigurálása során, meg kell határozni, milyen primitíveket használjon a rendszer. A VK_PRIMITIVE_TOPOLOGY_TRIANGLE_LIST beállításával meghatároztam, hogy a renderelés háromszögeket használjon alap primitívként. A Viewport és Scissor beállításával, meg kellett határozni a képernyőn megjelenő kép méretét és tartományát.

Ezen kívül be kellett állítani az olyan grafikus csővezeték paramétereit, mint a Viewport mérete és kezdeti koordinátája, a framebuffer minimális és maximális mélysége, illetve a viewport mélységi értékei. Ezek azt határozták meg, hogy a renderelt tartalom milyen mélységi tartományban érvényes. Például a minimális mélység, ami az a sík, amelynél közelebbi objektumok már nem láthatók. Maximális mélység (depth max) ami pedig a távolsági határ, amely után az objektumok már nem lesznek megjelenítve.

4.2. Renderelési fázis (Render Pass) és alfázisai (Subpasses)

A grafikus csővezeték kimenetének kezeléséhez szükség van egy Render Pass-ra. A Render Pass feladata, hogy kezelje a csővezeték kimeneti adatait és meghatározza, hogy hány és milyen framebuffer formátumokat fog használni, illetve lehetővé tegye az alfázisok használatát, amelyek különböző feldolgozási lépéseket végezhetnek. Az alfázisok mindegyike tud egy saját Graphics Pipelint tartalmazni. Ennek ellenére a projektemben nem használtam több alfázist mert rendundánsak lettek volna.

Így látszik, hogy a Render Pass hierarchikus. Ennek a hierarchiának a tetején van a Render Pass ami a teljes renderelési folyamat fő szakasza, Subpasses amik az egyes részfolyamatokat külön szakaszokra bonthatják, a Graphics Pipelines amin belül pedig a Shaders vannak.

Ez a rendszer lehetővé teszi, hogy egyetlen renderelési cikluson belül több különböző feldolgozási lépést végezzünk el, például egy árnyékolási passzt vagy egy posztprocesszáló effektet. Most már csak vissza kell kapni a GPU-tól az adatokat.

A Framebuffer egy olyan memóriaterület, amelybe a GPU a renderelt képkockát írja. Amikor a Render Pass elkészít egy képet, azt a kijelölt Framebufferhez továbbítja. A Framebufferhez egy vagy több képkocka kapcsolódhat, amelyekbe a Render Pass kimeneti adatai kerülnek. Ezzel a rendszer képes arra, hogy a grafikus csővezeték által generált képkockákat megfelelően tárolja és később felhasználja.

A Framebuffer biztosítja, hogy a renderelési folyamatnak legyen egy kijelölt célmemóriája, ahol az eredményeket tárolhatjuk. Azonban ez önmagában még nem elegendő, mert szükség van egy olyan mechanizmusra is, amely lehetővé teszi a parancsok küldését a GPU-nak és biztosítja, hogy minden művelet megfelelő sorrendben történjen.

4.3. GPU parancsok végrehajtása

A GPU-nak szánt Vulkan parancsokat először egy megfelelő adatstruktúrában kellett eltárolnom. Erre szolgál a Command Buffer, amely tartalmazza a renderelési és egyéb grafikus parancsokat, például a grafikus csővezeték beállítását végző vkCmdBindPipeline hívást vagy a rajzolási parancsokat kiadó vkCmdDraw hívást. A Command Buffer azonban nem hozható létre közvetlenül, először egy Command Pool-t kellett létrehozni.

A Command Pool a Command Bufferek memóriakezeléséért felelős. Ez egy memóriaterületet foglal a GPU-n, amelyből a Command Buffereket lehet allokálni. A Command Pool hatékonyabbá teszi a memóriakezelést, mert lehetővé teszi, hogy a Command

Buffer-ek ne közvetlenül a rendszermemóriából, hanem egy előre lefoglalt memóriaterületről jöjjenek létre. Amikor egy új parancsra van szükség, a Vulkan a Command Pool-ból oszt ki egy megfelelő méretű blokkot.

A Command Buffer-ek elkészülte után a parancsokat egy Queue (végrehajtási sor) kapja meg. A Queue szerepe, ahogy az korábban már említésre került, hogy a parancsokat végrehajtásra továbbítsa a GPU számára. A Command Buffer tehát nem fut le azonnal, hanem előbb a Queue-ba kerül, amely sorban végrehajtja a benne lévő műveleteket. Ez biztosítja, hogy a parancsok megfelelő sorrendben és hatékonyan fussanak le a GPU-n.

4.4. Szinkronizálás, zászlók (Flage) és Szemaforok (Semaphore)

A kezdeti implementáció során egy problémába ütköztem. A program nem tudta pontosan megállapítani, hogy egy kép elkészült-e vagy sem, emiatt előfordulhatott, hogy a rendszer még egy képre rajzolt, miközben azt már éppen a képernyőre próbáltam kirakni. Ez villódzást és hibás megjelenítést eredményezett.

Ennek a problémának a megoldására Semaphore-t használtam. A Semaphore egy olyan jelzőzászló, amelyet a GPU belső szinkronizációs műveleteire lehet használni. Ennek segítségével biztosítani tudtam, hogy a renderelési folyamat minden lépése megfelelő sorrendben fusson le. Amikor a GPU befejezte a renderelést, a Semaphore jelzi, hogy az adott kép készen áll a további feldolgozásra.

A másik probléma az volt, hogy a végrehajtási sor folyamatosan egyre több memóriát kezdett használni. Ezt a feladatkezelőn keresztül vettem észre. A GPU végrehajtási sora gyorsabban kapta a grafikus utasításokat, mint amilyen gyorsan azokat végre tudta hajtani. Emiatt a sor folyamatosan növekedett, és egyre több memóriát foglalt el.

Ennek a problémának a kezelésére Fence-eket (kerítéseket) alkalmaztam. A Fence egy olyan CPU és GPU közötti használható jelzőzászló, amely lehetővé teszi a CPU számára, hogy várakozzon a GPU-ra, amíg az be nem fejezi a kijelölt műveleteket. Ha a Fence nincs beállítva, a CPU addig nem küld újabb utasítást, amíg a GPU nem jelez, hogy befejezte az előzőt. Ezzel megakadályoztam, hogy a GPU végrehajtási sora túltöltődjön, és így elkerültem a memóriafogyasztás folyamatos növekedését.

A rendszer működésének optimalizálásához elengedhetetlen volt ezeknek a szinkronizációs mechanizmusoknak a beállítása. A Semaphores biztosította, hogy az egyes renderelési lépések megfelelő sorrendben fussanak le, míg a Fence gondoskodott arról, hogy a CPU ne adjon ki túl sok utasítást a GPU számára.

Ezeknek az elemeknek az összehangolt működése biztosította, hogy a Vulkan renderelési folyamata megfelelően működjön, és a képkockák gördülékenyen jelenjenek meg a képernyőn. A Framebuffer tárolta a renderelési eredményt, a Command Buffer kezelte a GPU-nak szánt utasításokat, a Queue gondoskodott a parancsok végrehajtásáról, míg a Semaphores és a Fence biztosította a megfelelő szinkronizációt és a memóriahatékony működést. Mindezek elkészítésével, a projektem gerince elkészült.

4.5. Vertex adatok shaderhez küldése

Most, hogy már rendelkeztem egy működő renderelő eszközzel, el kellett érnem, hogy az pontosan azt renderelje, amit szeretnék, és ne egy előre beégetett adathalmazt. Ehhez biztosítanom kellett, hogy a csővezetékem képes legyen külső adatokat fogadni és feldolgozni, például vertex adatokat.

A Vertex Buffer egy olyan GPU memóriában lévő adatszerkezet, amely háromdimenziós objektumok csúcspontjait (vertexeit) tárolja. Mivel a Vulkan nem rendelkezik beépített geometriai primitívekkel, ezért minden rajzolási művelet előtt az adatokat egy Vertex Buffer-ben kell elhelyezni, amelyet a Vertex Shader képes feldolgozni. Emellett szükség volt egy Device Memory-ra, amely a ténylegesen használatra szánt vertex adatokat reprezentálja. Ezt a memóriát a heapből allokálja a Vulkan, és az eszközmemória megfelelő típusa kulcsfontosságú a teljesítmény és az adathozzáférés szempontjából.

A VK_MEMORY_DEVICE_LOCAL_BIT egy olyan memória, amely kifejezetten a GPU számára van optimalizálva. Ez azonban nem érhető el közvetlenül a CPU számára, és csak Command Bufferek segítségével lehet rá adatot írni. Ezzel szemben a VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT típusú memória már elérhető a CPU számára, így az adatok közvetlenül feltölthetők. Az optimális teljesítmény érdekében mindenképpen az előbbi megoldást akartam használni, azonban ez nem volt közvetlenül elérhető a CPU számára. Egy megkerülő megoldásra volt szükségem.

A probléma megoldására két különálló puffert hoztam létre Az egyik HOST_VISIBLE, amelyet a CPU tud írni, a másik pedig DEVICE_LOCAL, amelyet a GPU gyorsan tud kezelni. Egy Command Buffer segítségével az adatokat a CPU számára elérhető memóriából átmásoltam a GPU számára optimalizált memóriába. Miután ez a két puffer létrejött, össze kellett őket kapcsolnom, és biztosítanom kellett, hogy a vertex adatok megfelelő módon elérhetők és kezelhetők legyenek.

Ezen a ponton azonban egy újabb problémával szembesültem. Az így kialakított struktúra rendkívül sok felesleges memóriát foglalt, különösen összetettebb objektumok

esetén. Ennek az volt az oka, hogy egy objektum csúcspontjai több élhez is tartozhatnak. Egy téglatest például két háromszögből állhat össze, azonban ezek a háromszögek osztoznak egy közös oldalon, vagyis ugyanazokat a csúcspontokat többször is el kellene tárolni, ami felesleges memóriahasználathoz vezet.

A probléma elkerülése érdekében Index Buffert használtam, amely lehetővé teszi, hogy egy vertexet többször is fel lehessen használni egy előre definiált egyedi vertexeket tartalmazó listából. Ezáltal minimalizálható a redundáns adattárolás, mivel ahelyett, hogy ugyanazt a csúcspontot többször is tárolnánk, elég csak egyetlen példányban eltárolni, és az indexek segítségével hivatkozni rájuk.

Mielőtt azonban ez a rendszer teljes mértékben működőképes lett volna, frissítenem kellett a Vertex Shader-t, hogy ne előre beégetett vertexekkel dolgozzon, hanem azokat a futásidőben kapott külső adatokból olvassa be. Ezzel a megoldással sikerült elérnem, hogy a grafikus csővezeték dinamikusan tudja kezelni a kívánt objektumokat, miközben optimalizált memóriafelhasználással dolgozik.

4.6. Adatküldés a shadereknek

A bufferek használata azonban nem az egyetlen módja annak, hogy adatokat továbbítsunk a shader számára. Egy másik megoldás az Uniform Buffer, amely egy olyan GPU memóriaterület, amelyet a shader kizárólag olvasásra használhat. Ez a módszer elsősorban globális, a teljes renderelési ciklus alatt állandó adatok tárolására alkalmas.

Általában transzformációs mátrixok, például a modell-, a nézeti- és a projekciós mátrixok továbbítására használjuk. A saját implementációmban is ezeknek az adatoknak az átadására vettem igénybe ezt a módszert, mivel ezek a paraméterek a renderelési folyamat során nem változnak minden egyes objektum esetén. Az Uniform Buffer egyik nagy előnye, hogy a GPU gyorsítótárazza ezeket az adatokat, így az olvasásuk rendkívül hatékony. Mivel ezek a mátrixok és egyéb beállítások több shader számára is hasznosak lehetnek, egyetlen uniform buffer segítségével akár az egész renderelési pipeline számára elérhetővé tehetők.

Azonban ezek az előnyök bizonyos hátrányokkal is járnak. Az egyik legnagyobb korlátozás, hogy az Uniform Buffer maximális mérete Vulkanban körülbelül 64 KB (bár ez függ a GPU implementációjától). Ha nagyobb adatmennyiséget kellene tárolni, akkor más megoldást kell keresni. Egy másik jelentős hátrány, hogy az Uniform Buffer tartalma a shaderből nem módosítható, tehát kizárólag a CPU tudja frissíteni az adatokat, ami egyesesetekben rugalmatlanságot eredményezhet.

A másik adatátviteli módszer, amelyet használtam, a Push Constant. Ez egy rendkívül gyors és hatékony módja annak, hogy a CPU közvetlenül, extra memóriafoglalás nélkül továbbítson kis mennyiségű adatot a GPU számára. A Push Constants egyik legnagyobb előnye, hogy nem igényel buffer objektumokat vagy descriptor seteket, ami minimális CPU overheadet jelent. Az adatok közvetlenül a renderelési parancsokkal együtt továbbíthatók, így nincs szükség külön memóriafoglalásra vagy másolásra, és a deszkriptorok frissítése is elkerülhető.

Felmerülhet a kérdés, hogy ha ez a módszer ennyire hatékony, akkor miért nem használtam minden adat továbbítására. A válasz abban rejlik, hogy a Push Constants legnagyobb előnye egyben a legnagyobb hátránya is. A Vulkan specifikáció szerint a maximálisan továbbítható adat mérete 128 byte, ami rendkívül korlátozott, különösen akkor, ha komplexebb adatokat kellene kezelni. Ez azt jelenti, hogy a Push Constants kizárólag kis méretű, gyorsan változó paraméterek továbbítására alkalmas, például egy objektum egyéni beállításaira, skálázási faktorára vagy effektusokra. Ha ettől nagyobb méretű adatokat kell átadni, akkor mindenképpen Uniform Bufferre vagy más adatkezelési mechanizmusra van szükség.

A két módszer tehát különböző célokra ideális. Az Uniform Buffer kiváló választás, ha nagyobb mennyiségű adatot kell kezelni, amely ritkán változik, míg a Push Constants gyors, minimális overhead-del járó módszer, amely akkor hasznos, ha gyakran változó, kis méretű adatokat kell átadni a shader számára. Egy hatékony Vulkan alkalmazás általában mindkét megoldást kombinálja, hogy optimális teljesítményt érjen el a renderelési folyamat során.

4.7. Depth Buffer

A programom így már képes volt egyetlen komplex objektum értelmezésére és betöltésére, de mi történik akkor, ha egyszerre több objektumot szeretnék betölteni? Ekkor találkoztam egy érdekes hibával. A probléma abból adódott, hogy amikor egy objektumot elkezdtem felrajzolni, a következő objektum mindig felülírta az előzőt a rajzolási folyamat során. Ennek eredményeként attól függően, hogy melyik objektum volt közelebb a kamerához, az lett később kirajzolva.

Ez azért jelentett problémát, mert ha két objektum takarja egymást, és a távolabbi objektum kerül később kirajzolásra, akkor az a közelebbi objektum előtt jelenik meg, ami nyilvánvalóan helytelen. A probléma megoldására egy Depth Buffer-t (mélységi puffer) használtam.

A Depth Buffer célja, hogy az objektumok kamerától való távolságát nyomon kövesse. A mögötte álló logika az, hogy ha egy adott pixelhez tartozó fragment mélyebben van (távolabb a kamerától), mint az előzőleg kirajzolt fragment, akkor annak a háttérben kell maradnia. A Depth Buffer ezt az információt egy külön képként (image) tárolja, amely minden egyes pixelhez egy mélységi értéket rendel.

Ennek megfelelően létrehoztam egy új képet, amely a mélységi adatokat tárolja, de mivel a Swapchain nem támogatja a mélységi képek létrehozását, ezért ezt manuálisan kellett megoldanom. Ehhez a VkImageCreateInfo struktúrát használtam a mélységi kép inicializálására, majd Device Memory-t kellett lefoglalnom, amelyhez ezt a képet hozzárendeltem. Ez az új mélységi kép hasonlóan működik, mint a Swapchain által létrehozott képek, azonban ezt a Render Pass egy alárendelt erőforrásaként kellett konfigurálnom.

A mélységi kép nézetét ezt követően a Framebufferhez rendeltem hozzá, hogy a csővezeték megfelelően kezelje azt. Ezen kívül néhány apróbb változtatást kellett végrehajtanom a grafikus csővezeték beállításaiban. A legfontosabb módosítások közé tartozott a mélységi tesztelés engedélyezése, valamint a Depth Buffer felülírásának beállítása, hogy a mélységi értékek helyesen frissüljenek minden egyes rajzolási ciklus során.

4.8. Textúrázás

Most, hogy a program már képes volt egy időben több komplex objektum megjelenítésére, egy újabb problémával kellett szembenéznem: minden objektum statikus, fekete felszínnel rendelkezett. Ez azt jelentette, hogy a program még nem támogatta a textúrák kezelését, így ennek a megoldása lett a következő célom.

A Vulkanban a textúráknak két alapvető része van: a kép (image) és a mintavevő (sampler). A kép (VkImage) maga a GPU memóriájában tárolt nyers textúraadat, amelyet a shader képes feldolgozni. A mintavevő (VkSampler) pedig egy olyan objektum, amely meghatározza, hogyan történjen a textúra mintázása, például milyen interpolációs módokat használjon, hogyan történjen a mipmapping, és milyen szűrési algoritmusokat alkalmazzon.

A mintavételező létrehozása viszonylag egyszerű a VkSamplerCreateInfo struktúra segítségével. Ez az objektum beállítja a szűrési módokat (például bilineáris vagy trilineáris szűrés), az anisotrópikus szűrést és a wrap módokat, amelyek meghatározzák, hogy a textúra hogyan ismétlődjön a széleken.

Mielőtt azonban elkezdhettem volna dolgozni a textúrákkal, először be kellett tölteni azokat. Erre a célra az stb_image külső könyvtárat használtam, amely lehetővé tette a JPEG,

PNG és egyéb formátumok könnyű beolvasását. Miután a képadatokat beolvastam a CPU memóriájába, azokat egy bufferbe kellett másolni, amelyből később a GPU számára is elérhetővé váltak.

A textúra adatokat azonban nem lehet közvetlenül a GPU-ba másolni, mert a VkImage objektumok optimalizált GPU-memóriát használnak, amelyet a CPU nem érhet el közvetlenül. Ezért egy Staging Buffer használatára volt szükség. A Staging Buffer egy átmeneti memória, amely lehetővé teszi, hogy a CPU először egy CPU által elérhető memóriába töltse be a textúrát, majd onnan másoljuk azt egy VkImage objektumba, amelyet a GPU optimalizált memóriában tárol.

A másolás után azonban a textúra még nem használható közvetlenül a shaderben, mert alapértelmezés szerint a memóriaformátuma nem megfelelő a mintavételezéshez. Ahhoz, hogy a shader helyesen tudja olvasni az adatokat, a Pipeline Barrier mechanizmusát kellett alkalmaznom.

A Pipeline Barrier biztosítja, hogy a textúraadatok a megfelelő memóriaállapotba kerüljenek, és garantálja, hogy az egyes műveletek befejeződjenek, mielőtt a következő feldolgozási lépés megtörténik.

A megfelelő állapotváltás után már csak az volt hátra, hogy a shader számára elérhetővé tegyem a textúrát. Vulkanban a shaderek nem közvetlenül férnek hozzá a textúrákhoz, hanem egy Descriptor Set segítségével kell azt elérhetővé tenni.

A Descriptor Set egy olyan struktúra, amely lehetővé teszi, hogy a shader dinamikusan hozzáférjen a textúrához és a mintavevőhöz, anélkül hogy azokat közvetlenül a shader kódjába kellene beégetni. A textúrát és a sampler objektumot a Descriptor Sethez kellett csatolni, amely biztosítja a shader számára, hogy a megfelelő adatokhoz férjen hozzá futásidőben.

A Descriptor Set beállítása során jelezni kellett, hogy a fragment shader fogja használni az adott textúra mintavevő kombinációt. Ehhez egy VkDescriptorImageInfo struktúrát kellett létrehozni, amely összekapcsolja a VkImageView és VkSampler objektumokat, majd ezt hozzárendeltem a descriptor bindinghoz. Ezután sikeresen át tudtam adni a textúra és textúra mintázó (sampler) adatokat a shadernek ahol egy beépített metódus segítségével az objektumomat textúrázni tudtam.

5. KÜLSŐ OBJEKTUMOK BEOLVASÁSA

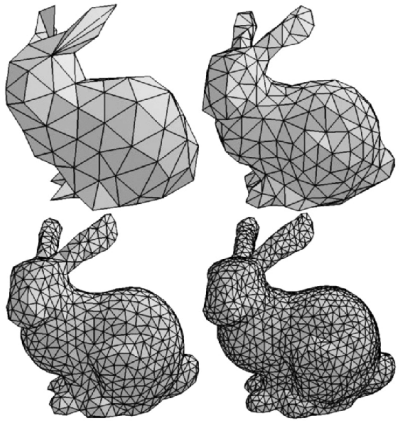
5.1. Hálókezelő osztály

Most már a programom készen állt arra, hogy külső modelleket, például Blenderrel készített 3D modelleket használjon. Ehhez egy újabb külső könyvtárat hívtam segítségül: az Open Asset Importer Library-t, vagy röviden Assimpot.

Azonban mielőtt ennek nekikezdhettem volna, könnyebben kezelhetővé kellett tennem a modellek kezelését. Eddig a pontig, a modellek mindig manuálisan voltak beégetve a kódba, azonban most, külső objektumok kezelésére készültem. Ezért szükségem volt egy kifinomultabb megoldásra. Erre a célra létrehoztam két osztályt. A Mesh és a MeshModel osztályt, amelyek felelősek a modellek és modellhálók kezeléséért, valamint azok memóriakezeléséért.

A Mesh osztály a legalapvetőbb renderelési egységet képviseli. A mesh egy háromdimenziós modellháló, ami csúcsokból áll, amelyeket indexek határoznak meg. A Mesh osztály ezen adatokat Vulkan buffer objektumokba tölti, és tartalmazza az ehhez szükséges logikát. A vertex- és indexbufferek létrehozását, valamint azok memóriakezelését.

A konstruktora paraméterként megkapja a szükséges Vulkan-erőforrásokat, mint például a logikai és fizikai eszközöket, illetve a renderelendő csúcsokat és indexeket. Ezekből létrehozza a szükséges Vulkan puffereket, amelyek a GPU-n tárolják az adatokat. A texId mező lehetővé teszi, hogy az adott hálóhoz egy adott textúra legyen kötve egy azonosító segítségével, így lehetővé téve, hogy több háló esetén is mindig tudni lehessen, hogy melyik hálóhoz melyik textúra tartozik.



ábra 5.1
Mesh model

A MeshModel osztály egy magasabb szintű absztrakciót nyújt. Egy MeshModel, ami egy objektumot képvisel, több Mesh objektumot tud egyszerre kezelni, ezáltal egy teljes 3D modellt reprezentál. Például egy emberi karakter külön hálókból állhat, mint a fej, test és végtagok, amelyek mindegyike külön Mesh objektumként létezhet. A MeshModel ezeket egyetlen egységként kezeli, és a modell transzformációs mátrixán keresztül pozicionálható a világban.

A MeshModel tartalmaz pozíció és irányinformációt, valamint egy controlable attribútumot, amellyel jelölhető, hogy a modell felhasználói interakcióval mozgatható vagy nem. A keyControl metódus segítségével egy egyszerű billentyűzetes irányítást valósítottam meg, amellyel lehetővé tettem az objektumok térbeli mozgatását. Ezek elkészítésével a programom készen állt a külső objektumok beolvasását kezelő elem beillesztésére.

5.2. Adat beolvasás

Az Assimp egy hatékony modellbetöltő könyvtár, amely lehetővé teszi különböző 3D fájlformátumok beolvasását és feldolgozását egy egységes adatstruktúrába.

A modellek beolvasásához a createMeshModel függvényt készítettem, amely egy fájlnév alapján importálta a modellt, majd az azt alkotó hálók (mesh-ek) és textúrák feldolgozását is elvégezte. A hálók általános kezeléséhez egy külön osztályt hoztam létre, amely a mesh-ek struktúráját és adatait kezelte.

A Mesh (háló) egy 3D modell alapvető építőköve, amely geometriai információkat tartalmaz. Egy mesh csúcspontokból (vertexekből), háromszögekből (face-ekből), normálvektorokból és textúrakoordinátákból áll. Fontos megjegyezni, hogy a textúrafájl neve meg kell egyezzen az OBJ fájlban hivatkozott névvel, különben a program nem tudja megfelelően betölteni azokat.

Az Assimp segítségével történő modellimportálás során néhány előfeldolgozási beállítást is meg kellett adnom, amelyek segítettek a modell megfelelő kezelésében.

A aiProcess_Triangulate enum az összes poligont háromszögekre bontotta, a aiProcess_FlipUVs a textúrakoordináták helyes tükrözését biztosította és a aiProcess_JoinIdenticalVertices az azonos csúcspontokat egyesítette, csökkentve ezzel a szükségtelen adatduplikációt és optimalizálva a modell feldolgozását.

Miután a modell betöltése megtörtént, a következő lépés az objektumok és textúrák feldolgozása volt. Az Assimp segítségével beolvastam az objektumhoz tartozó textúrákat, majd ezeket egy listában tároltam.

Ezt követően a textúrák listáját átalakítottam egy textúraindex-listává, amely összekapcsolta az anyagokat a shaderben használt descriptor set megfelelő indexeivel.

Ez a folyamat lehetővé tette, hogy minden egyes anyaghoz egyedi textúrát rendeljek, és biztosította, hogy az egyes hálók a megfelelő textúrát használják a renderelés során.

A textúrák megfelelő feldolgozása után a modell geometriáját is létre kellett hozni. A következő lépésben a program a betöltött modell hálóit (mesh-eket) dolgozta fel, majd azokat egy új 3D objektumként hozta létre.

A végső lépésben egy MeshModel objektumot hoztam létre, amely tartalmazta a modell összes hálóját, majd a modellt egy listába mentettem, hogy az elérhető legyen a renderelési folyamat során.

Ezzel a program képessé vált külső modellek kezelésére és megjelenítésére a Vulkan-alapú renderelőmotoromban.

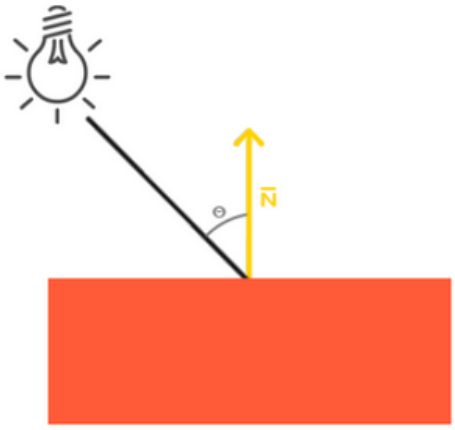
6. FÉNYEK KEZELÉSE ÉS A PHONG-FÉLE VILÁGÍTÁSI MODELL

A projektem ekkor már közel állt a befejezéshez, azonban még szükség volt a megfelelő fénykezelés implementálására. A programomban három különböző fénytípust hoztam létre, Ambient, diffuse és specular fényt, amelyek együtt alkotják a Phong-féle megvilágítási modellt. Ez egy olyan világítási rendszer, amely egyszerű, de mégis elegendően valósághű hatást biztosít a számítógépes grafikában.

A valóságban a fény viselkedése rendkívül összetett, számos tényező befolyásolja, például a fény visszaverődése, törése, szóródása vagy az árnyékok keletkezése. Egy számítógépes program esetében azonban korlátozottak az erőforrások, így olyan modelleket kell alkalmazni, amelyek egyszerűbbek, de mégis hatékonyan képesek szimulálni a valós világban tapasztalható fényhatásokat. A Phong-modell az egyik legismertebb megvilágítási technika, amely különböző összetevők segítségével próbálja megközelíteni a valósághű fényviselkedést.

Az ambient fény azt szimulálja, hogy a valóságban soha nincs teljes sötétség. Mindig jelen van egy alapvető háttérvilágítás, amelyet például a környezetben visszaverődő fények, a hold fénye vagy a távoli fényforrások hoznak létre. Ennek a célja, hogy az objektumoknak mindig legyen látható színük, még akkor is, ha éppen nem éri őket közvetlen fényforrás. A legegyszerűbb megközelítés az ambient fény kezelésére egy globális konstans érték alkalmazása, amely minden felületre egy minimális világosságot biztosít. Bár léteznek ennél jóval összetettebb módszerek, például a globális megvilágítási algoritmusok, amelyek figyelembe veszik a fény visszaverődési tulajdonságait, ezek jelentősen növelnék a számítási költséget, és a projekt keretein kívül esnek.

A diffúz fény az irányított fényeket modellezi, és azt határozza meg, hogy egy adott felület mennyi fényt kap a fényforrás irányából. Az alapelve, hogy minél inkább a fényforrás felé fordul egy felület, annál intenzívebb a megvilágítása. Ha egy felület merőlegesen áll a fény irányára, akkor teljes fényerőt kap míg, ha a fény laposan éri a felületet, akkor az kevésbé világos. Ezt az összefüggést a Lambert-féle fényvisszaverődési modell írja le, amely szerint a fény intenzitása a normálvektor és a fény irányvektorának szögétől függ. Ennek megfelelően a fény irányát és az objektum normálvektorát felhasználva kiszámolható, hogy az adott felület mennyire kap fényt.



6.1 ábra
Diffuse fény

A speculáris fény a diffúz megvilágításhoz hasonlóan a fény irányától és az objektum normálvektorától függ, azonban egy további tényezőt is figyelembe kell venni: a nézőpontot. A spekuláris fény azt a jelenséget modellezi, amikor egy fényes felület egy pontban erősen visszaveri a fényt, és az a megfelelő szögben elhelyezkedő szemlélő számára egy fényes kiemelkedő pontként jelenik meg. Ezt a hatást úgy értem el, hogy kiszámoltam a fényvisszaverődési irányvektort a felület normálvektora alapján, majd ezt összehasonlítottam a kamera irányával. Minél közelebb esik a két vektor egymáshoz, annál erősebb a spekuláris hatás, amely egy fényes pontot eredményez a felületen.

A fények számításához szükséges adatokat egy struktúrán belül kezeltem, és a fragment shader számára a Uniform Buffer Object (UBO) segítségével továbbítottam. Mivel az UBO egy olyan memóriaegység, amelyet a GPU olvasásra használhat, így ideális megoldást biztosított az olyan adatokhoz, amelyek nem változnak minden egyes renderelési ciklusban, mint a fény adatok. Az UBO segítségével továbbítottam a fényforrás pozícióját, az irányvektort, a fény színét és intenzitását, valamint az egyes fénykomponensek erősségét a shader számára.

A fényhatásokat tovább finomítottam egy spotlight effekttel, amely lehetővé tette, hogy a fényforrás ne egy minden irányba sugárzó pontfény legyen, hanem egy meghatározott irányba világító kúpos fényforrás. Ehhez meg kellett határozni a fény kúpjának belső és külső határát, amely szabályozta, hogy milyen szögben érje el a fény az objektumokat. A spotlight hatás biztosította, hogy az adott fényforrás csak egy bizonyos irányba és szögben világítson, így létre tudtam hozni egy zseblámpához a megfelelő fényt.

A világítás természetesebb megjelenítése érdekében távolságfüggő csillapítást is alkalmaztam. A valóságban a fény intenzitása csökken a távolsággal, így ennek modellezésére

egy matematikai formulát használtam, amely biztosította, hogy a fény távolodva egyre halványabb legyen. Bár ennek pontos paraméterezése függ a jelenet méretarányától és az alkalmazott fényforrás típusától, az implementált megoldás biztosította a fények természetesebb viselkedését. Ezzel teljessé téve a projektemet.

7. ÖSSZEGZÉS

Így összegezve:

- Sikeresen felállítottam egy Vulkan-alapú renderelő motort, amely képes komplex 3D objektumok valós idejű megjelenítésére és mozgatására.
- A grafikus csővezeték minden fő elemét implementáltam (Vertex Shader, Fragment Shader, Render Pass, Framebuffer, stb.).
- Megvalósítottam külső modellek betöltését az Assimp könyvtár segítségével.
- Kialakítottam egy kamerarendszert, amely támogatja a valós idejű nézőpontváltást billentyűzettel és egérrel.
- Megvalósítottam a Phong-féle világítási modellt, beleértve az ambient, diffuse és specular fénykezelést.
- Textúrázást valósítottam meg Vulkanban, több textúra egyidejű használatával.

A fejlesztés során mélyreható tapasztalatokat szereztem a Vulkan API működéséről, a grafikus csővezeték felépítéséről, az erőforrások kezeléséről, valamint a renderelési folyamat különböző szakaszairól. Külön figyelmet fordítottam a memóriahatékony adattárolásra, a több objektummal való egyidejű kezelésre, illetve a renderelési teljesítmény optimalizálására. A rendszer képes valós idejű kamerakezelésre, világításszimulációra, és támogatja az interaktív objektumvezérlést is.

A projekt jelenlegi állapotában már alkalmas külső modellek betöltésére és megjelenítésére. Ugyanakkor ez még nem tekinthető véglegesnek vagy tökéletesnek. Ennek ellenére a most elkészült rendszer szilárd alapot nyújt a további fejlesztéseimnek, mint például árnyékok, tükröződések vagy akár animációk integrálása. Amikor ezek megvalósulnak, a motor ideálisan kiegészíthető lesz egy fizikai motorral, vagy akár hangkezelő alrendszerrel is.

Hiszem, hogy amíg valami nem tökéletes, addig mindig van lehetőség a javításra és a továbbfejlesztésre. Még ha a tökéletesség nem is érhető el teljes mértékben, a projektemmel az ideál felé való törekvés önmagában is értékes és ösztönző cél.

IRODALOMJEGYZÉK

[1] Learn OpenGL. Graphics Programming. Írta: Joey de Vries
[2] Vulkan Tutorial. Legutolsó látogatás: 2025.03.20
[3] Kamera definiálál ábra: Learn OpenGL
[4] Mailbox prezentációs mód ábra: Legutolsó látogatás: 2025.03.20
[5] Vulkan Grafikus csővezeték ábra: Legutolsó látogatás: 2025.03.20
[6] Mesh model ábra: Legutolsó látogatás: 2025.04.02
[7] Diffuse fény ábra: Learn OpenGL

Vulkan renderelő motor

NYILATKOZAT

Alulírott Horváth Levente Programtervező Informatikus Bsc szakos hallgató, kijelentem, hogy a dolgozatomat a Szegedi Tudományegyetem, Informatikai Intézet Szoftverfejlesztés Tanszékén készítettem, programtervező informatikus Bsc diploma megszerzése érdekében.

Kijelentem, hogy a dolgozatot más szakon korábban nem védtem meg, saját munkám eredménye, és csak a hivatkozott forrásokat (szakirodalom, eszközök stb.) használtam fel. Tudomásul veszem, hogy szakdolgozatomat a Szegedi Tudományegyetem Diplomamunka Repozitóriumában tárolja.

2025. május. 10. Horváth Levente

KÖSZÖNET NYILVÁNÍTÁS

Ezúton szeretném kifejezni hálámat és köszönetemet Dr. Kiss Ákosnak, akinek útmutatásával, értékes tanácsaival és folyamatos támogatásával hozzájárult a munkám sikeréhez.

Továbbá, köszönettel tartozom szüleimnek és nővéremnek, akik biztatásukkal és kitartó támogatásukkal mindig mellettem álltak, és segítettek abban, hogy ezt az akkádját is sikerrel vegyem.

Hálával gondolok barátaimra, akik végig mellettem álltak, bíztattak a nehezebb pillanatokban és mindig emlékeztettek arra, hogy ha én nem is, de ők hisznek bennem. Az ő támogatásuk és biztatásuk nélkül ez az út sokkal nehezebb lett volna.

MELLÉKLETEK

Implementáció github-on.