



Budapesti Műszaki és Gazdaságtudományi Egyetem

Villamosmérnöki és Informatikai Kar

Irányítástechnika és Informatika Tanszék

Magyar Anna Zsuzsanna

ÁRNYÉK MEGJELENÍTÉSI TECHNIKÁK ELEMZÉSE

KONZULENS

Fridvalszky András Máté

BUDAPEST, 2023.12.07.

Tartalomjegyzék

ÖSSZEOFGLALÓ	5
ABSTRACT	6
1 BEVEZETÉS	7
1.1 A DOLGOZAT SZERKEZETE	7
2 A PROGRAM FELÉPÍTÉSE.....	8
2.1 A VULKAN RENDERER ALAPJAI.....	8
2.2 3D MEGJELENÍTÉS.....	10
2.3 RAJZOLÁS.....	11
2.4 KAMERA	12
2.5 DOMBORZAT.....	12
2.6 FÉNYFORRÁS	13
2.7 ÁRNYÉKTÉRKÉPEK	13
2.8 FELHASZNÁLÓI FELÜLET.....	13
3 IMPLEMENTÁLT TECHNOLÓGIÁK	14
3.1 ÁRNYALÁS	14
3.2 NORMAL MAPPING	15
3.3 MIPMAPPING.....	16
3.4 MULTISAMPLING	16
3.5 GAUSS-SZŰRŐ.....	17
4 ÁRNYÉKTÉRKÉP TECHNIKÁK.....	18
4.1 SIMPLE SHADOW MAPPING.....	18
4.1.1 Algoritmus.....	18
4.1.2 Mellékhatalások	19
4.1.3 Implementáció.....	21
4.2 PERCENTAGE CLOSER FILTERING	22
4.2.1 Algoritmus.....	22
4.2.2 Implementáció.....	22
4.3 CASCADED SHADOW MAPPING	23
4.3.1 Algoritmus.....	23
4.3.2 Mellékhatalások	23
4.3.3 Implementáció.....	24
4.4 VARIANCE SHADOW MAPPING	27
4.4.1 Algoritmus.....	27
4.4.2 Mellékhatalások	28
4.4.3 Implementáció.....	30
4.5 EXPONENTIAL SHADOW MAPPING	31

<i>4.5.1 Algoritmus</i>	31
<i>4.5.2 Mellékhatások</i>	32
<i>4.5.3 Implementáció</i>	32
5 EREDMÉNYEK	33
5.1 KOMBINÁCIÓS LEHETŐSÉGEK	33
5.2 TELJESÍTMÉNYMÉRÉS	35
6 ÖSSZEGZÉS	38
IRODALOMJEGYZÉK	39

HALLGATÓI NYILATKOZAT

Alulírott **Magyar Anna Zsuzsanna**, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot/ diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettettem, egyértelműen, a forrás megadásával megjelöltetem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2023. 12. 07.

.....
Magyar Anna Zsuzsanna

Összefoglaló

Az árnyékok szerepe és jelentősége a valósághű grafikai megjelenítésben kiemelkedő fontosságú. Valós időben, dinamikusan változó környezetben az árnyékok hiteles megjelenítése komoly számítási kapacitást igényel, ami hatékony algoritmusok alkalmazását követeli meg. Számos technika létezik dinamikus árnyékok megvalósítására, viszont a legtöbbet használt és megfelelő eredményeket adó technika az árnyéktérképezés. A módszernek az évek során különböző változatai alakultak ki, melyek egyre inkább a valóságot tükrözik.

A szakdolgozatom keretében egy Vulkan API-ra épülő grafikus megjelenítőt valósítottam meg C++ nyelven. A valós idejű realisztikus megjelenítés érdekében számos különböző technikát implementáltam, de a hangsúly a különböző árnyék leképezési módszerekre irányult, azon belül is az árnyéktérképezésre. A különböző árnyéktérkép technikák részletes bemutatása mellett ismertetem az implementált megjelenítő felépítését, majd teljesítmény szempontjából értékelem az eredményeket.

Abstract

The role and significance of shadows in realistic graphic representation are highly important. Rendering shadows authentically in real-time and in dynamically changing environments requires significant computational capacity, demanding the use of efficient algorithms. There are many techniques for implementing dynamic shadows, but the most commonly used technique that gives good results is shadow mapping. Over the years, various versions of this method have developed, which increasingly reflect reality.

Within the scope of my thesis, I implemented a renderer based on the Vulkan API using C++. To achieve real-time realistic rendering, I implemented several different techniques, but the focus was on different shadow rendering methods, within shadow mapping. In addition to a detailed presentation of the various shadow mapping techniques, I present the structure of the implemented renderer and then evaluate the results in terms of performance.

1 Bevezetés

A számítógépes grafika napjainkban mindenütt jelen van. A videójátékok világai, a filmek látványos effektusai, és a digitális tervezés általunk használt eszközei mind a számítógépes grafika eredményei. A technológia fejlődése és a számítógépek növekvő teljesítménye lehetővé teszi az egyre realisztikusabb megjelenést.

A játékfejlesztés az egyik terület, amire a 3D számítógépes grafika lenyűgöző hatást gyakorol. A videojátékok egyre nagyobb elvárásoknak kell megfeleljenek, és a játékfejlesztők folyamatosan törekednek a valósághű és részletes virtuális világok létrehozására. Ebben a kontextusban fontos szerepet játszanak a grafikai programozási interfések, amelyek lehetővé teszik a kommunikációt a hardver és a szoftver között.

A Vulkan API [1] egy alacsony szintű, platformfüggetlen grafikus és számítási felület, amelyet a Khronos Group fejlesztett. Első kiadása 2016-ban jelent meg az OpenGL utódjaként, azzal a céllal, hogy egy hatékonyabb és modernebb megközelítést biztosítson a grafikai programozás terén. A Vulkan-t arra tervezték, hogy teljes mértékben kihasználja a hardverek képességeit, ezzel lehetővé téve a teljes renderelési folyamat irányítását a fejlesztők számára.

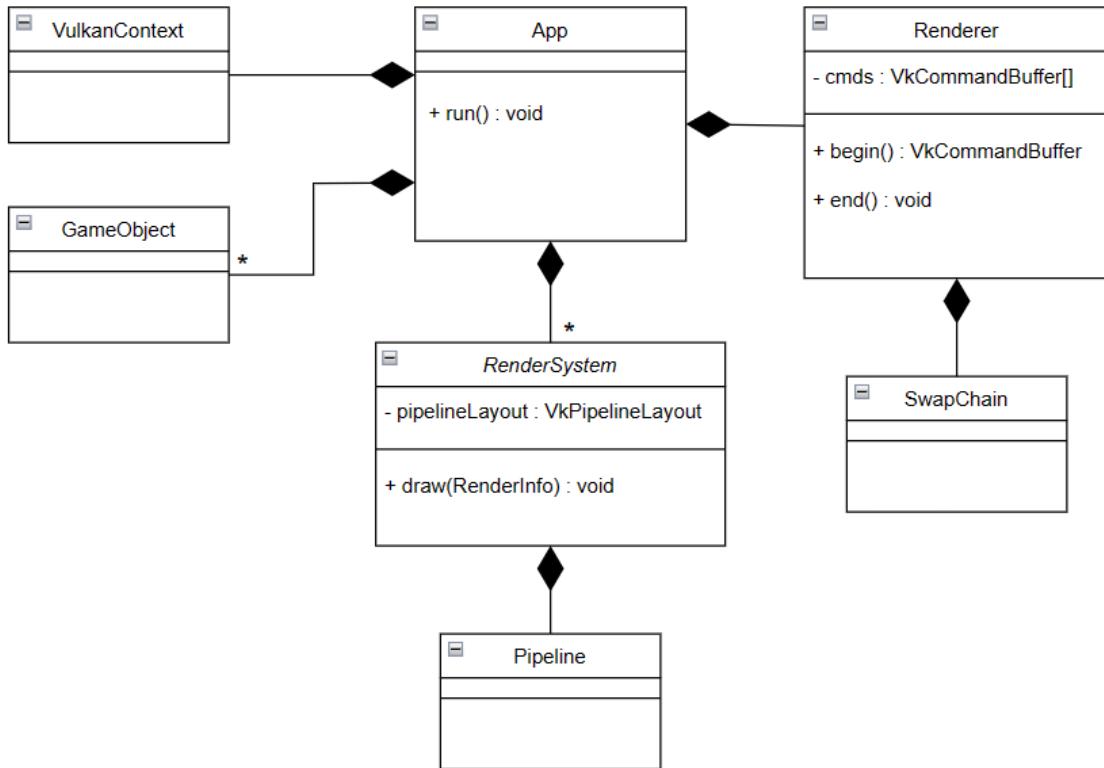
1.1 A dolgozat szerkezete

A következő fejezetben bemutatom az általam készített Vulkan API-ra épülő megjelenítő program felépítését. A leírást a Vulkan API építőköveivel kezdem, amelyek feltétlenül szükségesek a grafikai megjelenítéshez. Ezt követően bemutatom azokat az osztályokat, amelyek lehetővé teszik a 3D megjelenítést, valamint a rajzolás folyamatát. Ezen felül az árnyéktérképezés algoritmusok által megkövetelt és a tesztelésben segítséget nyújtó osztályokat ismertetem.

A 3. fejezet a különféle általános árnyalási technikákat fogja tárgyalni, árnyalási, textúrázási módszereket és az árnyéktérképezés technikákhoz köthető és felhasználható technológiákat. A 4. fejezetben pedig részletesen bemutatom a legelterjedtebb árnyéktérképezési módszereket, mind algoritmus mind pedig implementáció szempontjából. Ez magába foglalja a különböző módszerek részletes leírását és ezeknek a Vulkan API-ra történő alkalmazását.

2 A program felépítése

Ebben a fejezetben bemutatom a megjelenítő program alapját képező fontosabb osztályokat és azok kapcsolatait (2.1. ábra), illetve az árnyéktérkép technikák implementálásához szükséges osztályokat.



2.1. ábra Fontosabb osztályok kapcsolatai

2.1 A Vulkan renderer alapjai

A Vulkan nem biztosít közvetlen ablakkezelő integrációt, ezért a GLFW [17] könyvárat használtam az ablakok létrehozására és a bemeneti események kezelésére. A **Window** osztály feladata az ablak létrehozása, amit egy Vulkan felülethez (`VkSurface`) rendeltem, beállítva azt megjelenítési felületnek.

Az **Instance** osztály inicializálja a Vulkan könyvtárat egy példány (`VkInstance`) létrehozásával, ami az alkalmazással való kapcsolatot reprezentálja. A helyes API használat érdekében beállítottam a Vulkan beépített hibakezelési rétegeit is.

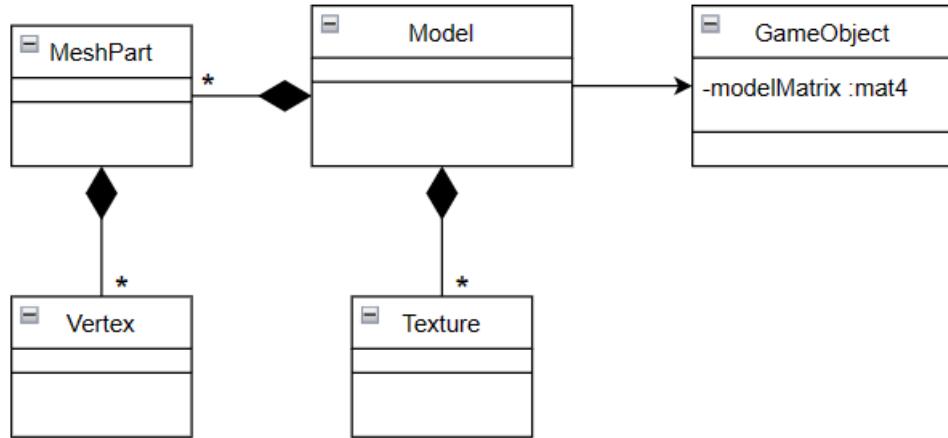
A **Device** osztály felelőssége a megfelelő grafikus kártya kiválasztása, amely támogatja a szükséges szolgáltatásokat. Ellenőrizni kell a queue families (sor családok) és swapchain támogatását, mivel nem minden grafikus kártya képes a képeket közvetlenül a képernyőn megjeleníteni. A Vulkan-ban a legtöbb művelet, mint a rajzoló parancsok és a memória műveletek, aszinkron hajtódnak végre egy VkQueue sorban, amelyeket sor családokból allokálunk. A megfelelő hardver kiválasztása után, a Device osztály létrehoz egy logikai eszközt (VkDevice), amely a fizikai eszköz interfészét képviseli. A logikai eszköz erőforrások lefoglalására, command bufferek létrehozására és egyéb műveletek végrehajtására szolgál.

A Vulkan API-ban a swapchain egy olyan rendszer, ami tartalmazza a képernyőn megjelenítésre váró képek sorát. Általános célja a képek megjelenítésének szinkronizálása a képernyő frissítés gyakoriságával. Rajzolás során a swapchain-ből lekért képekre rajzolunk, majd visszahelyezzük őket a megjelenítéshez. A **SwapChain** osztály felelőssége tárolni magát a swapchain-t (VkSwapchainKHR) és a hozzá tartozó erőforrásokat. A swapchainben lévő képeket (VkImage) egy framebufferben (VkFrameBuffer) tároljuk, amihez egy kompatibilis képnézzel (VkImageView) csatoljuk. Az osztály feladata a swapchain-hez tartozó szinkronizálási eszközök, a VkSemaphore és a VkFence tárolása is. A semaphore a GPU-n történő műveletek végrehajtási sorrendjének meghatározására szolgál, a fence pedig a CPU-t és GPU-t tartja szinkronban.

A **Pipeline** osztály feladata létrehozni a grafikus csővezetéket (VkPipeline), a névtérben deklarált *ConfigInfo* struktúra segítségével, ami tartalmazza a szükséges beállításokat. Mivel a program több Pipeline-t tartalmaz, létrehoztam egy *createDefaultConfigInfo* metódust, ami visszaad egy alapértelmezett pipeline konfigurációt, így a különböző Pipeline-ok definiálásánál csak módosítani kell az alapértelmezett beállítás megfelelő részeit létrehozás előtt.

2.2 3D megjelenítés

A 3D objektumok tárolásához használt osztályokat az alábbi ábra szemlélteti.



2.2. ábra Az objektumok kapcsolata a modellel

A csúcspontok tulajdonságának definiálására egy **Vertex** struktúrát hoztam létre, ami a következőket tárolja: koordináta, szín, textúra-koordináta, normál vektor és tangens vektor.

A **MeshPart** a modell egy részét reprezentálja. Tárolja a modell egy részének *Vertex* tömbjét, indexeit és a hozzá tartozó textúra indexét. A struktúra tárolja ezen kívül a csúcspontokhoz és indexekhez tartozó buffereket. A Vulkan bufferei olyan memória területek, amelyeket a grafikus kártya által olvasható adatok tárolására használunk.

A **Model** osztály tárol egy MeshPart típusú tömböt és a hozzájuk tartozó textúrák tömbjét. A wavefront obj [22] kiterjesztésű modellek betöltésére a tinyobjloader [18] könyvtárat használom, majd végig iterálva az alakzatokon feltöltöm a megfelelő vektorokat. A modell betöltése után létrehozom a shaderekbe továbbítandó buffereket és definiálom a grafikus csővezetékben az attribútum és binding leírásokat.

A **GameObject** osztály reprezentálja az egyes objektumokat a programban, amelyek tartalmaznak egy referenciát az aktuális objektum Model-jére. Az osztály határozza meg a model mátrixot, ami az objektum elhelyezkedésének tulajdonságait írja le. A programban használt mátrix-okat a GLM [19] könyvár segítségével hozom létre. Mivel a model mátrix folyamatosan változhat a renderelés során, ezért uniform bufferként töltöm fel a shaderbe. Ezeket a buffereket leírók (descriptor set-ek) segítségével lehet hozzáférhetővé tenni a shader számára. A descriptor set-ek létrehozásához először

definiálni kell azok descriptor set layout-ját, ami leírja a descriptor set típusát és amit meg kell adni a Pipeline létrehozásakor. Majd ezeket a leírókat egy descriptor pool-ból kell allokálni. minden rajzoláskor frissítem a uniform buffereket majd használat előtt bindolom a hozzájuk tartozó descriptor set-eket.

A **Textura** osztály meghatározza egy darab MeshPart textúráit. Ebben az esetben egy diffúz- és normál map textúráról beszélünk. A textúrák elérési útvonalát az objektum fájljában definiált mtl fájlból olvasom be és a képek betöltését az stb [20] könyvtárral végzem. A diffúz textúra esetében mipmapokat generálok a teljesítmény optimalizálása érdekében és textúra mintavételezésénél alkalmazom a többmintás élsimítást (MSAA). A shaderhez való csatoláshoz egy descriptor set-et hozok létre, amihez csatolom a textúra képnézetét és mintavételezőjét.

2.3 Rajzolás

A **Renderer** osztály felelőssége tárolni a SwapChain-t és a rajzoláshoz szükséges command buffereket. A Vulkan parancsai, mint például a rajzolási műveletek és a memóriaátvitelek, nem közvetlenül függvényhívásokkal hajthatók végre. Az összes végrehajtani kívánt műveletet rögzíteni kell egy command buffer objektumban. Ezeket a buffereket egy command poolból kell allokálni, amely kezeli a tárolásukra használt memóriát. A Renderer osztály *begin* metódusa lekéri a swapchain következő képét, amely a renderelés célja lesz, és visszatérési értékként adja azt a command buffert, amiben majd a rajzolási parancsokat rögzítjük. A renderelés végeztével meghívom az *end* metódust, amiben a rögzített parancsokat végrehajtom és megjelenítem.

A programban különféle *renderSystem* osztályokat definiáltam, amelyek a **RenderSystem** abstract osztályból származnak. A *renderSystem* osztályok egyetlen publikus metódusa a rajzolás, ezen kívül létrehozzák és tárolják a rajzolásban használt Pipeline-t és VkPipelineLayout-ot. A kód duplikáció elkerülése érdekében a grafikus csővezeték létrehozásánál a *renderSystem*-ek az alapértelmezett pipeline konfigurációt egészítik ki, vagy módosítják. A rajzolásban résztvevő objektumokat és a renderpass-t paraméterként egy *RenderInfo* típusú struktúrában kapják.

A **Main_RenderSystem** osztály felelőssége a képernyőre való renderelés, a többi offscreen *renderSystem* a paraméterként kapott textúrába rajzol a tulajdonságaihoz mérten.

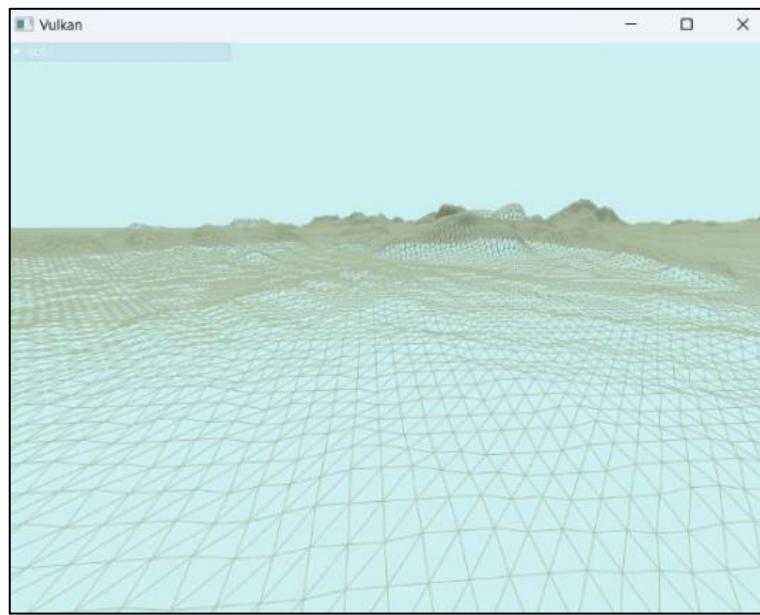
A gameloop-ot az **App** osztály *run* metódusában definiáltam, amiben első lépésként a uniform bufferek frissítése történik, majd a parancsok rögzítése és végrehajtása a megfelelő *Renderer* és *renderSystem* hívásokkal.

2.4 Kamera

A könnyebb tesztelhetőség érdekében létrehoztam egy perspektív kamera-modellt, amit a WASD, shift, space billentyűkkel és az egérrel irányíthatunk. A **Camera** osztály tartalmazza a szem pozícióját és irányát. Felelőssége definiálni és minden rajzolás előtt frissíteni, a nézeti- és projekciós mátrixot, amiket későbbiekben a shaderekbe továbbítok uniform változóként.

2.5 Domborzat

Az árnyékok megjelenítésére szükség volt egy padló szerű objektumra, ezért létrehoztam egy **Terrain** osztályt, ami egy domborzatot reprezentál és önmagában elegendő az árnyékok teszteléséhez. A terrain létrehozásához a height map [3] technikát alkalmaztam, vagyis egy textúrából töltöm fel a domborzat csúcspontjainak magasságát. A domborzat kirajzolásához egy új Pipeline-t és pipeline layout-ot definiáltam a Main_RenderSystem-ben.



2.3. ábra Terrain

2.6 Fényforrás

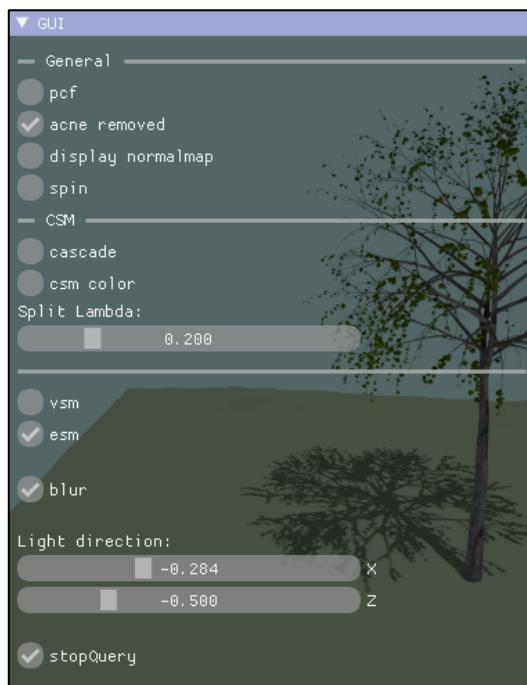
A **Light** osztály reprezentálja a programban definiált irányfényforrást. Az irányfényforrások mindenütt azonos irányú és intenzitású fényt sugároznak, ezért a fény projekciós mátrixát egy ortografikus mátrix-szal hoztam létre.

2.7 Árnyéktérképek

Az árnyéktérképek erőforrásait külön **Shadowmap** osztályba szerveztem, amiből polimorfizmus segítségével további specifikus árnyéktérkép osztályokat hoztam létre. Az implementálásnál a legfontosabb szempont, hogy egy olyan framebuffert kell létrehozni az árnyéktérképnek, amibe először is renderelünk, majd később textúraként mintavételezünk, a színtér kirajzolása során.

2.8 Felhasználói felület

A program dinamikus teszteléséhez implementáltam egy felhasználói felületet, a Dear ImGui [21] könyvtár segítségével. Ezáltal a program futása alatt változtathatóak az egyes paraméterek és rajzolási technikák. A **Gui** osztály inicializálja az ImGui könyvtárat és minden képkockában frissíti az ablakot.



2.4. ábra Felhasználói felület

3 Implementált technológiák

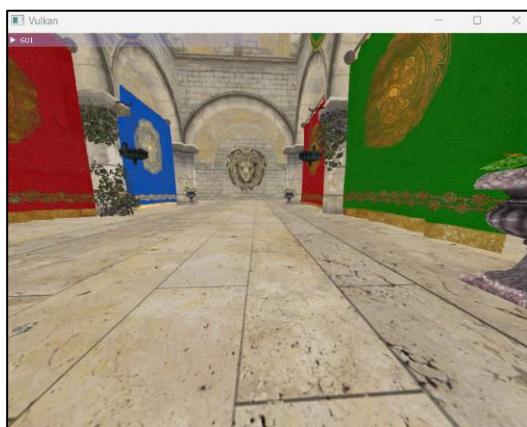
Ebben a részben a teszt környezet realisztikus megjelenítése érdekében implementált- illetve az árnyéktérkép algoritmusoknál felhasznált technológiákat mutatom be.

3.1 Árnyalás

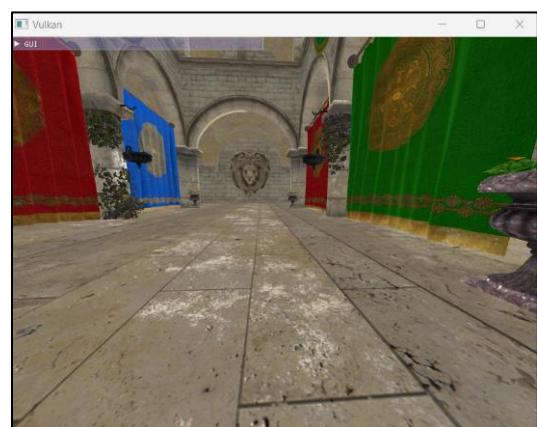
A színtér árnyalásához az ambiens (környezeti) és diffúz megvilágítási modellt alkalmaztam.

A fény sugarak többszöri visszaverődést követően elérhetnek olyan helyeket is, amelyek közvetlenül nem láthatóak a fényforrásból. A globális megvilágításra az ambiens megvilágítási tényezőt a fragmensek színértékéhez adtam, így úgy tűnhet, mintha minden lenne valamennyi szort fény a jelenetben.

A diffúz megvilágítás már szembe tűnőbb változásokat eredményez, mivel az árnyalást az objektumokra eső fény sugarának szögéből képezi. Ha a fény sugarának merőleges az objektum felületére, akkor van a legnagyobb hatása a fénynek. A fény sugarának szögét az adott pont normál vektorának és a fény sugarának normalizált koordinátájának skalárszorzatával számoltam.



3.1. ábra Árnyalás nélküli megjelenítés

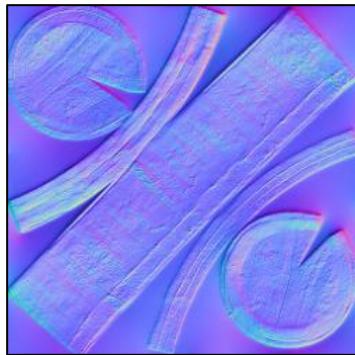


3.2. ábra Ambiens és diffúz árnyalás
normal mapping technikával

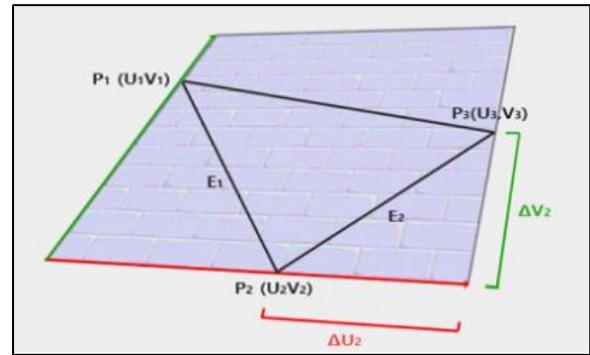
3.2 Normal Mapping

A háromszögekből álló modellekre burkolt textúrák növelik a valóságszerű megjelenést, de ha alaposabban megnézzük, meglehetősen könnyű látni a textúrák alatt lévő síkfelületeket. Világítás szempontjából az egyik módja annak, hogy egy tárgy alakját látszólag megváltoztassuk, az a felület normál vektorához köthető. A normal mapping [4] technika a felületenkénti normál vektor helyett, a fragmensek normál vektorát egy textúrából (3.3. ábra) mintavételezi. Ezáltal a megvilágított objektumok összetettebbnek tűnnek.

A probléma az, hogy ha változik az objektum orientációja, a normal map értékei többé már nem lesznek relevánsak. A megoldás a **tangent space**: ahol a normál vektorok mindenkor a pozitív z irány felé mutatnak és a többi megvilágítási vektort ehhez a pozitív z irányhoz képest átalakítjuk. A megvilágítási vektorok transzformálását a **TBN** mátrix inverzával tesszük, amit az adott háromszög tangens-, bitangens- és normál vektorából készítünk.



3.3. ábra Normal map



3.4. ábra A tangens (piros) és bitangens (zöld) vektor ábrázolása [3]

Az egyes csúcspontokhoz tartozó tangens- és bitangens vektorok így képezhetőek le:

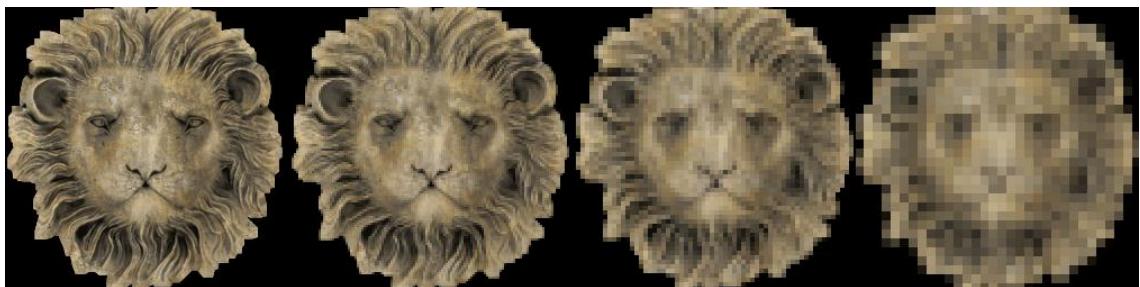
$$\begin{bmatrix} T \\ B \end{bmatrix} = \begin{bmatrix} \Delta U_1 & \Delta V_1 \\ \Delta U_2 & \Delta V_2 \end{bmatrix}^{-1} \cdot \begin{bmatrix} E_1 \\ E_2 \end{bmatrix}$$

A módszer implementálásához a csúcspontok tulajdonságait kiegészítettem azok tangens vektorával, majd a színtér kirajzolásához felhasznált vertex-shaderben kiszámoltam a TBN mátrix inverzét és transzformáltam a megvilágítási vektorokat.

3.3 Mipmapping

A mipmap-ok egy textúra előre kiszámított kicsinyített változatai (3.5. ábra). A módszer arra épül, hogy a távolabbi objektumok esetében nincs szükségünk a nagy részletességre, így azoknál használhatunk alacsonyabb felbontású textúrákat. A mipmap-ok használatával optimalizálhatjuk a teljesítményt anélkül, hogy romlana a megjelenés minősége.

Generálásuk Vulkan környezetben teljesen testre szabható. A különböző méretű textúrákat a VkImage képünk mip szintjein tároljuk, a mintavételezőben pedig beállíthatjuk a mip szintek tartományát. [2]



3.5. ábra A különböző mip szintek textúrái

3.4 Multisampling

A többmintás élsimítás (multisample anti-aliasing) egy népszerű módszer a textúra éleinél keletkező kockásodás megszüntetésére. Az átlagos renderelés során a pixel színe egyetlen mintavételi pont alapján lesz meghatározva, ami általában a pixel közepén található. Ha a rajzolt vonal egy bizonyos pixelt érint, de nem fedi le a mintavételi pontot, az adott pixel üresen maradhat, ami az éles lépcsőzős hatást eredményezi. Az MSAA technika több mintavételi pontot használ pixelenként, hogy meghatározza a végső színt. [2]

A módszer implementálásához átalakítottam a swapchain képek framebuffer-ét, hogy a pixeleket először egy offscreen képbe rendereljem, így a kép az MSAA technikával mintavételezve kerül majd megjelenítésre. Ezt követően a hardver kapacitásának megfelelően beállítottam a grafikus csővezeték rászterizálási folyamatában, hogy mennyi mintavételezési pontot használjon a rajzolás során.

3.5 Gauss-szűrő

A Gauss-elmosás félárnyékok létrehozására is szolgál olyan árnyéktérképezési módszereknél, ahol a mélységértékeket lineárisan szűrhető módon reprezentáljuk.

Ez a kép-elmosási eljárás egy adott pont környezetének átlagolásával csökkenti a zajt és az éles élek hatását. A módszer [6] a Gauss-eloszlás két változós sűrűségfüggvényét használja origó középponttal, a súlyozott átlagolásra. A várható érték ebben az esetben az eloszlás középpontját jelenti, így a használt függvény így alakul:

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}},$$

ahol σ^2 az eloszlás szórásnégyzete.

A kernel súlyok meghatározásához kiértékelhetjük a Gauss-függvényt, vagy használhatjuk a Pascal-háromszöget is, mivel a Gauss-eloszlás diszkrét megfelelője a binomiális eloszlás.



3.6. ábra Elmosás nélküli variancia
árnyéktérkép



3.7. ábra A gauss-elmosás eredménye

4 Árnyéktérkép technikák

A realisztikus megjelenítés egyik legfontosabb feladata az árnyékok implementálása. Az árnyékok jelentősen növelik a mélységérzetet a megvilágított jeleneteinkben és megkönnyítik a tárgyak közti térbeli kapcsolatok érzékelését.

A két legelterjedtebb árnyékolási algoritmus valós idejű alkalmazásokban a shadow mapping [7] és a shadow volumes [8]. A kettő közül az árnyéktérkép technikák népszerűbbek, mivel számos előnyük van az árnyéktérképezéshez képest; például könnyebben implementálhatóak, és a költségük kevésbé érzékeny a geometriai bonyolultságra. Napjainkban a ray tracing [9] is használható valós időben a megfelelő hardveres támogatással. A sugárkövetés előnye a pontosság, ugyanakkor nagy teljesítményt igényel.

A legismertebb árnyéktérképezési algoritmusok közül egy felosztásos (CSM) illetve két lineárisan szűrhető (VSM, ESM) árnyéktérképezési módszert implementáltam, az általános árnyéktérképezésen (SSM) kívül.

4.1 Simple Shadow Mapping

Árnyékok akkor keletkeznek, amikor egy tárgy elzárja a fény útját, így a mögötte lévő blokkolt objektum árnyékba kerül. Egy fényforrás, egy adott irányban csak egyetlen objektumot tud közvetlenül megvilágítani. Tehát az, hogy egy pont árnyékban van-e, meghatározható a pont fényforrástól mért távolságának és az ugyanezen irányban található esetlegesen blokkoló objektum fényforrástól mért távolságának összehasonlításával. Az általános árnyéktérképezés (SSM) módszere erre a megfigyelésre épül.

4.1.1 Algoritmus

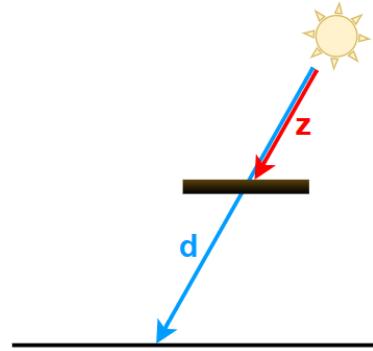
Az algoritmus két fázisból áll. Az első fázisban a színtér mélység koordinátáit egy textúrába rendereljük a fényforrás szemszögéből. Ezáltal a textúra, amit árnyéktérképnek is nevezünk a fényforráshoz legközelebb eső pontok mélységét tárolja. Ennek kivitelezéséhez a fényforrásunk tulajdonságaiból adódó nézeti- és projekciós mátrixszal transzformáljuk az adott pontunk pozícióját.

A második fázisban, a színteret újból kirajzoljuk, összehasonlítva a kameránk által látott pontok fényforrástól mért távolságát a texturából kiolvasott mélységekkel. Ezt az összehasonlítást az árnyékteszt alkalmazásával végezzük el.

Az árnyékteszt a következőképpen definiált:

$$s(x) := f(d(x), z(p)) := \begin{cases} 1, & z(p) \geq d(x) \\ 0, & z(p) < d(x) \end{cases}$$

$x \in R^3$ az a pont, amiről elakarjuk dönten, hogy árnyékban van-e, $p \in R^2$ pedig ennek a pontnak a lightspace-beli megfelelője. A $d(x)$ jelöli a pontunk távolságát a fényforrástól, míg a $z(p)$ ugyanezen az irányon a fényforrástól mért leghamarabbi ütközési pont távolságát jelenti. Azaz a $z(p)$ az árnyéktérkép p -vel mintavételezett értéke.

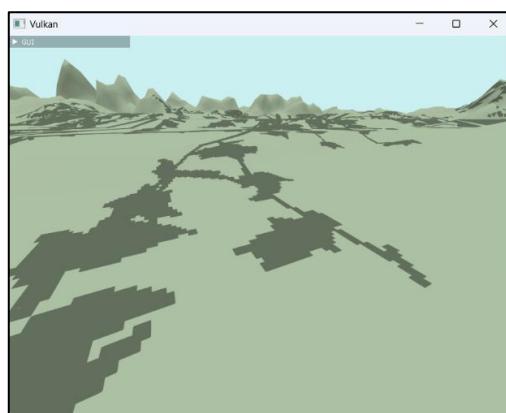


4.1. ábra Az árnyék képződése

Tehát, ha az árnyéktérképből kiolvasott z mélységek kisebb, mint az aktuális pontunk fényhez viszonyított távolsága akkor pixelünk árnyékban lesz és a direkt megvilágítás értékét 0-val szorozzuk.

4.1.2 Mellékhatások

Az árnyéktérképezési technikával létrehozott árnyékok fő hátránya az aliasing, amely hatása az éles árnyékszélek és a felbontás okozta kockásodás (4.2. ábra). Ez annak köszönhető, hogy az árnyéktérkép gyakran alacsonyabb felbontása miatt, egy texelhez több képernyőpixel is tartozhat. A felbontás növelésével ezen tudunk javítani, de ez további számítási költséggel jár.



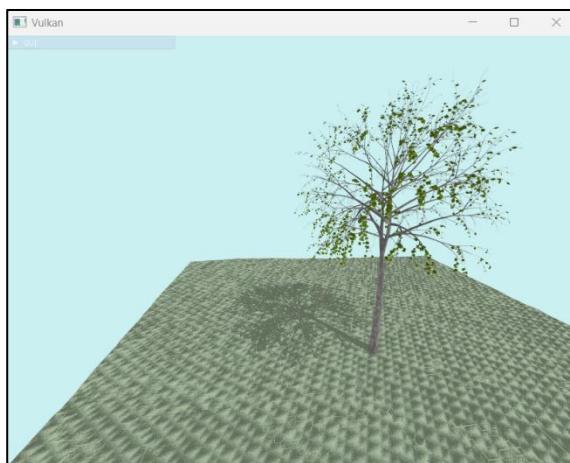
4.2. ábra Az alacsony felbontás okozta kockásodás

A másik megoldás az élsimítás lenne, de mivel mélységértékeket tárolunk azok elmosásával csak azt érnénk el, hogy az átlagolt mélységértékeket hasonlítanánk az aktuális pontunk mélységértékéhez, ami ugyanúgy 0-át vagy 1-et ad eredményül.

4.1.2.1 Shadow acne

A shadow acne jelenség pontossági problémák miatt jelentkező hibás önrnyékolást jelent. Ez akkor jelentkezik, amikor a diszkrét mintavételezés során az árnyék kiszámítása hibásan történik. Ez ugyancsak az árnyéktérkép alacsony felbontásának köszönhető, mivel ilyenkor előfordulhat, hogy néhány pixelnél hibásan kisebb mélységérték lesz eltárolva.

Ennek eredménye a szabályos csíkozott, zajos árnyék (4.3. ábra). Azok a felületek érintettek, amelyek viszonylag távol helyezkednek el a fényforrástól és alap esetben nincsenek árnyékban.



4.3. ábra Shadow acne

A probléma megoldását árnyékeltolásnak (biasing) nevezzük. Egyszerűen eltoljuk az árnyéktérképből mintavételezett mélységeket egy kis értékkal, így elkerülhetjük, hogy a fragmenseket tévesen árnyékban értelmezzük.

4.1.2.2 Peter panning

A peter panning jelenség az árnyékeltolás következménye. Akkor válik szembetűnővé, ha a shadow acne megoldásához használt eltolás (bias) értéke viszonylag nagy. A peter panning effektus olyan illúziót kelt, mintha az objektumok kissé elszakadnának az általuk vetett árnyéktól.

Ahhoz, hogy megoldjuk a jelenséget, a rasszterizálás során a cull mode-ot beállíthatjuk, hogy az objektumok elülső lapját vegye figyelembe mikor az árnyéktérképet készítjük.

Mivel az árnyéktérképben csak a mélység értékeket tároljuk, ezért egy tömör objektumnál nem számít, hogy az első lapját vagy a hátsót használjuk a mélység kiszámításához.

4.1.2.3 Textúrán kívüli mintavételezés

Egy másik vizuális probléma, hogy a fényforrás befoglaló dobozán-án kívül eső területeket árnyékban látjuk. Erre a megoldás az, hogy nullázzuk az árnyék mértékét, ha a pixelünk fénytérbe vetített koordinátája kívül esik ezen a területen, vagyis a mélység nagyobb mint 1. Ezen felül a mintavételezőn beállíthatjuk a címzési módot `VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_BORDER`-re, ami a textúrakoordináták értékeit a megfelelő [0,1] intervallumra szabályozza.

4.1.3 Implementáció

A szabványos árnyéktérképet reprezentáló **DepthShadowMap** osztály a *Shadowmap* egyik leszármazottja, framebufferében lévő kép egy egyszerű mélysékgép, amit 32-bites lebegőpontos mélység formátummal hoztam létre. A framebufferhez csatolás a kompatibilis képnézzettel történik.

A szabványos árnyéktérkép esetében csak a fény és modellek tulajdonságaival dolgozunk, így a pipeline layout-ban beállítottam a fény nézeti- és projekciós mátrixának, illetve az objektumok model mátrixának descriptor set layout-ját. Ezen felül a modell textúráit is feltöltöttem a fragment-shaderbe annak érdekében, hogy az objektumok átlátszó része az árnyéktérképből is törölve legyen. A pipeline konfigurációban a vertex-shader inputjaként csak a pozícióra és textúra koordinátára lesz szükség, így a többi *Vertex* struktúrában lévő tulajdonságot elhagytam.

A peter panning probléma kiküszöbölésére a cull mode-ot front-face-re állítottam és mivel csak mélységrétekkel dolgozunk a colorblendig folyamatot kikapcsoltam a pipeline-ban.

A vertex-shaderben történik a csúcspontok lightspace-be történő transzformálása, a fragment-shaderben pedig az átlátszóság ellenőrzése.

A második fázisban az árnyéktérképet a `Main_RenderSystem` osztályban definiált fragment-shaderbe olvasom és itt végzem el az árnyéktesztet.

4.2 Percentage Closer Filtering

A Percentage Closer Filtering (PCF) [10] az implementálást tekintve az egyik legegyeszerűbb mód lágy árnyékok létrehozására.

A normál színtextúrákkal ellentétben az árnyéktérképek textúráit, amik csak mélységinformációt tartalmaznak, nem lehet elő szűrni, így nem lehet rájuk használni az alapértelmezett textúra szűrő algoritmusokat. Ehelyett használhatjuk a PCF-et, ami kis mértékben növeli a kép minőségét azáltal, hogy hatására az árnyék szélek simábbnak tűnnek.

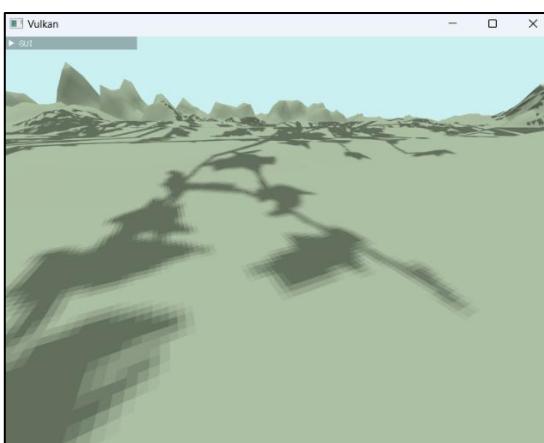
4.2.1 Algoritmus

A PCF algoritmus az árnyék mértékét az adott pixel környezetéből számolja. Tehát több mintavételezés történik fragmensenként, majd az eredmények átlaga adja az árnyék valószínűségét. Az algoritmus limitált formájára hardver támogatás is van, így az implementálás nagyon egyszerű.

4.2.2 Implementáció

A megvalósítás során az eredetileg Sampler2D mintavételezőt Sampler2DShadow típusuként töltjük fel a shaderbe. Ennek érdekében egy másfajta mintavételezőre van szükség, ami végrehajtja az árnyéktesztet is.

A mintavételezés ebben az esetben önmagába foglalja az aktuális pont, illetve környezetének árnyéktesztjét, majd ezeket átlagolja és visszatérési értékként adja. A beépített mintavételező kizárolag négy minta átlagolására működik, de ez általában elfogadható eredményeket szolgáltat, különösen akkor, ha a cél a költséghatékonyúság.



4.4. ábra 4x4-es PCF hatása

4.3 Cascaded Shadow Mapping

Az általános árnyéktérképezés egy viszonylag hatékony és egyszerű technika, de nem elegendő a jó minőségű árnyék előállításához összetett, nagyméretű jelenetekben. Ennek több oka is van:

- Nagy látótér esetén a mélység információk tárolására használt textúrák nem rendelkeznek elegendő felbontással.
- Az árnyéktérképek csak a fényforrás által megvilágított területet tartalmazzák, és nem azt a területet, amit a kamera aktuálisan néz.
- Előfordulhat, hogy a fényforrás projekciós mátrixa nem illeszkedik megfelelően a látóterünkre.

Növelte a textúra felbontását javíthatunk az árnyékok megjelenésén, de korlátoz minket a GPU memóriának kapacitása. A kaszkádolt árnyéktérképezés (CSM) [11] megoldást nyújt a felsorolt problémákra és növeli az árnyékok minőségét optimalizált memória-használat mellett.

4.3.1 Algoritmus

A módszer alapgondolata az, hogy nem szükséges ugyanannyi részletet alkalmaznunk a távoli objektumok árnyékolásánál, mint amennyire a közel objektumok esetében szükségünk van. Ennek érdekében a látóteret kaszkádokra bontjuk, és ezekre különböző részletekkel árnyéktérképeket készítünk.

Az algoritmus lépései:

- Látótér felosztása tetszőleges számú kaszkádra.
- A fényforrás nézeti- és projekciós mátrixának meghatározása minden kaszkádra.
- Az árnyéktérképek generálása.
- A színtér kirajzolása, mintavételezés a megfelelő árnyéktérképből.

4.3.2 Mellékhatások

A kaszkádolt árnyéktérképek egyik fő problémája a **light shimmering** (fény csillogás), ami hatására az árnyékok instabilnak és vibrálnak tűnnek. Ennek oka, hogy a kamera mozgásával a fényforrás nézeti- és projekciós mátrixa folyamatosan változik, így a leképzett árnyék texelek is változni fognak. A probléma megoldásához meg kell

határozni, hogy mekkora elmozdulásra van szükség a projekciós mátrixban, hogy az illeszkedjen az árnyéktérképhez. [13]

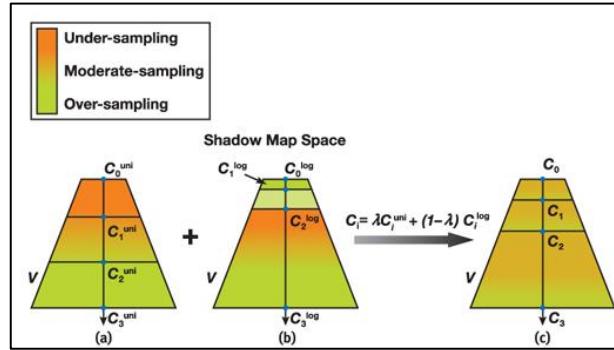
4.3.3 Implementáció

A **felosztási távolságok** kiszámítására a Parallel-Split Shadow Maps (PSSMs) [12] algoritmust használtam. A felosztási távolságokat kézzel is be lehet állítani, de ez a megközelítés optimális eredményeket ad.

A felosztási séma a következő egyenlettel írható le:

$$C_i = \lambda C_i^{\log} + (1 - \lambda) C_i^{\text{uni}} \quad 0 < \lambda < 1,$$

ahol C_i^{\log} és C_i^{uni} a logaritmikus és uniform felosztási sémából származnak.



4.5. ábra Az optimális felosztási séma [12]

A két felosztási séma a következő egyenletek szerint vezethető le. Ezeket egy állítható lambda értékkel súlyozva kombináljuk a felosztási távolságok meghatározása érdekében.

$$C_i^{\log} = n \left(\frac{f}{n} \right)^{\frac{i}{m}}$$

$$C_i^{\text{uni}} = n + (f - n) \frac{i}{m}$$

Az egyenletekben n a közeli síkot, f a távoli síkot reprezentálja, amíg i az aktuális kaszkád indexe, m pedig a kaszkádok számát jelenti.

A felosztási távolságok kiszámítása után, minden kaszkádra meg kell határozni annak **lightspace** mátrixát [13], ami a fényforrásnak az adott kaszkádra vonatkozó nézeti- és projekciós mátrixának szorzatát jelenti. Az alábbiakban egy tetszőleges kaszkád lightspace mátrixának definiálási folyamatát mutatom be.

Az ortografikus projekciós mátrixnak precízen illeszkednie kell a kaszkádunkra. Ehhez szükséges kiszámítanunk világterben az adott kaszkád sarokpontjait. A folyamat első lépéseként meghatároztam a teljes látótér világterbeli sarkait, amit úgy kapunk meg, hogy egy egységkockát transzformálunk a kameránk inverz nézeti- és inverz projekciós mátrixszával. Mivel a kameránk nézeti- és projekciós mátrixa végzi el azt a feladatot, hogy a látóterünk csonka kúpját NDC kockává alakítsa, a transzformáció megfordításával megkaphatjuk a látóterünk sarkait.

A látótér világterbeli sarkai közötti vektorok és az adott kaszkád felosztási távolsága segítségével már definiálhatóak a kaszkád sarokpontjai. A távoli sarkok új koordinátája egyenlő lesz a látótér megfelelő közelí sarkának koordinátája és az előbb kiszámolt irányvektor összegének az adott felosztási távolsággal vett szorzatával. A közelí sarkok koordinátája ehhez hasonlóan alakul, csak az előző kaszkád felosztási távolságát használjuk az egyenletben. Az így megkapott kaszkád távoli sarkai lesznek a következő kaszkád kiinduló pontjai.

A **projekciós mátrix** megalkotásának utolsó lépéseként meghatároztam az aktuális kaszkád középpontját és a középponttól a sarokpontokig számolandó maximális sugarat. Így a GLM [19] könyvtár felhasználásával megalkotható a mátrix, ami illeszkedik a kaszkádunk sarokpontjaira:

```
ortho(-radius.x, radius.x, -radius.y, radius.y, 0.0f, 2.0f * radius.z);
```

A fény **nézeti mátrixa** is könnyen meghatározható miután tudjuk a kaszkádunk középpontját:

```
lookAt(center - (lightDir * radius.z), center, vec3(0.0f, 1.0f, 0.0f));
```

A **light shimmering** elkerülése érdekében stabilizálni kell a projekciót. Ehhez először meghatároztam az árnyéktérkép középpontját az origó és a fényforrás nézeti- és projekciós mátrixának szorzatával, majd ezt igazítottam az árnyéktérkép méretéhez. A számított középpont és annak kerekített változatának különbségéből meghatároztam a szükséges eltolást és ezt az ortografikus mátrix eltolásához használtam. Ezzel a

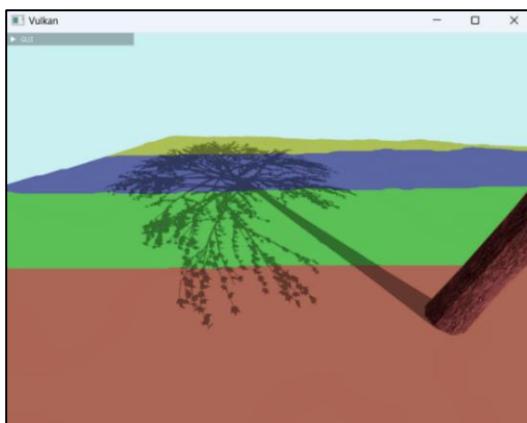
megoldással közelítőleg nullára redukáltam a light shimmering jelenséget. A következő kódrészlet mutatja az eltolás meghatározását:

```
vec4 shadowOrigin = lightSpaceMatrix * vec4(0.0f, 0.0f, 0.0f, 1.0f);
shadowOrigin = shadowOrigin * shadowMapSize / 2.0f;

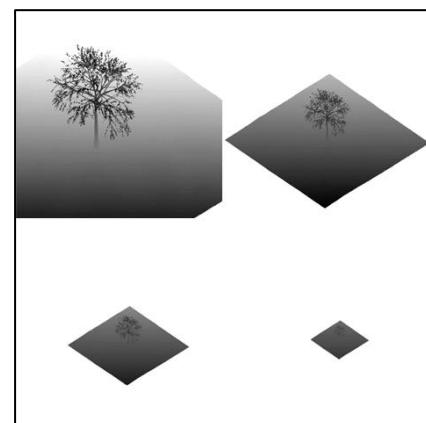
vec4 offset = round(shadowOrigin) - shadowOrigin;
offset = offset * 2.0f / shadowMapSize;
offset.z = 0.0f;
offset.w = 0.0f;
```

Az **árnyéktérképek generálásához** és eltárolásához egy új osztályt hoztam létre, ami a *ShadowMap* osztályból származik. Az osztályban tárolt mélysékgép *VkImageArray* típusú. Rétegei tartalmazzák a renderelés céljaként létrehozott framebuffer-eket. Ezáltal egységesen és könnyedén, indexeléssel használhatóak az egyes árnyéktérképek. A kaszkádokhoz tartozó lightspace mátrixoknak egy uniform buffert hoztam létre, amit descriptorok segítségével továbbítok a vertex-shaderbe. A renderelési folyamatot létrehoztam egy új *renderSystem* osztályt, aminek a rajzolási műveletében a paraméterként kapott árnyéktérkép egyes rétegeibe történik a renderelés. A rajzolási ciklusban a renderpassban beállított framebuffer az indexnek megfelelően változik. Az index shaderbe való továbbítására a Vulkan beépített push constants változóját használom, ami egy másik módja dinamikus értékek átadására. Ennél a megoldásnál nincs szükség uniform buffer-re, az értékeket rajzolás előtt állítom be.

Az algoritmus utolsó lépéseként a **színtér kirajzolása** következik. A fragment-shaderben kiszámolom, hogy az adott pixel melyik kaszkád szinthez tartozik, majd a fragmens-t a hozzá tartozó lightspace mátrixszal transzformálom, és a megfelelő árnyéktérképet mintavételezem vele. Ezt követően végrehajtom az általános árnyéktesztet.



4.6. ábra Kaszkádolt árnyéktérképezés



4.7. ábra A kaszkádokhoz tartozó árnyéktérképek

4.4 Variance Shadow Mapping

A szabványos árnyéktérképek egyik fő hátránya, hogy nem szűrhetők úgy, mint a színes textúrák, ami jellemzően súlyos aliasing-hoz vezet. Ezen probléma kiküszöbölésére alkalmazhatjuk a Percentage-Closer Filtering (PCF) algoritmust. Azonban a magas minőség elérése érdekében nagy mennyiséggű mintavételezésre van szükség, aminek eredménye a magas számítás igény.

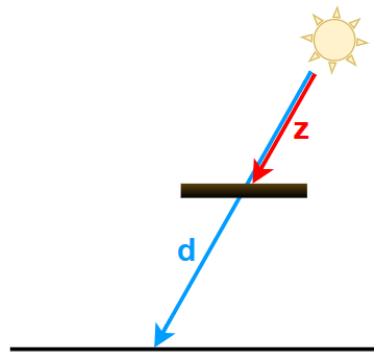
Egy másik megoldás az árnyéktérkép-szűrés problémájára a Variance Shadow Mapping (VSM) [14] technika. A fő gondolat, hogy a mélység adatokat lineárisan szűrhető módon ábrázoljuk, vagyis az árnyéktérkép értékei lineárisan kombinálhatóak lesznek. Ezáltal használhatjuk a modern GPU-k beépített algoritmusait, például a mipmapping-ot, és különböző textúra szűrési módokat. Így simább, pixelmentes árnyékszéleket kapunk.

4.4.1 Algoritmus

Az algoritmus valószínűség számítással határozza meg az árnyék mértékét. A PCF algoritmushoz hasonlóan arra vagyunk kíváncsiak, hogy hány százalékban van árnyékban az adott pixel, azaz mekkora annak a valószínűsége, hogy az árnyéktérképből különböző koordinátkkal kiolasott z értékek kisebbek, mint a rögzített d mélység. Ennek a valószínűségnek a megfordítása a Csebisev-egyenlőtlenség egyoldalú változata:

Legyen z egy valószínűségi változó valamelyen μ várható értékű és σ^2 szórásnégyzetű eloszlásból, ekkor, ha $d > \mu$:

$$P(z \geq d) \leq \frac{\sigma^2}{\sigma^2 + (d - \mu)^2} = P_{\max}$$



P_{\max} az árnyékoltság mértékének komplementere, amivel a későbbiekben a színértékeket szorozzuk a sötétítés érdekében.

4.8. ábra Az árnyék képződése

Az egyenlet nem ad pontos értéket a $P(z \geq d)$ -re, de ez sok esetben jó közelítés. [14]

Ahhoz, hogy meghatározhassuk ezt a valószínűséget az algoritmus nem csupán egyetlen mélységnégyzetet tárol az árnyéktérképben, hanem mélységnégyzetek (M_1) mellett tárolja azok négyzetét (M_2) is.

Annak érdekében, hogy az árnyéktérkép egy valószínűségi eloszlást reprezentáljon, el kell mosnunk a textúra értékeit. Ezt követően valamely fragmensre szűrve visszanyerjük a mélység-eloszlás M_1 és M_2 momentumait, amiből megkapjuk a várható értéket és a szórás négyzetet.

$$\mu = E(x) = M_1$$

$$\sigma^2 = E(x^2) - E(x)^2 = M_2 - M_1^2$$

Ennek tudatában már meghatározhatjuk a P_{\max} -ot, amivel az árnyékolás érdekében szorozzuk a pixelünk színét. A színtér kirajzolásánál, az árnyéktérkép mintavételezésénél a következő esetek merülhetnek fel:

- $d \leq \mu$: Az aktuális pixel mélységünk kisebb vagy egyenlő, mint a kiolvasott M_1 várható érték → a pixel nincsen árnyékban.
- $d > \mu$: a Csebisev-egyenlőtlenség képletét használva számítjuk az árnyék valószínűségét.

4.4.2 Mellékhatások

Az algoritmus fő problémája, hogy a Csebisev-egyenlőtlenség csak egy felső korlátot ad $P(z \geq d)$ valószínűségre, nem pedig egy pontos értéket.

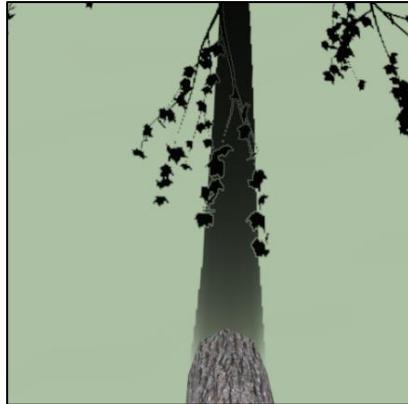
4.4.2.1 Light bleeding

A fényátszűrődés (4.9. ábra) jelenség azt eredményezi, hogy a fény olyan területeken jelenik meg, ahol a felületnek teljesen árnyékban kellene lennie. Ez általában akkor jelentkezik, amikor két árnyékoló objektum a fényhez viszonyítva egymás alatt helyezkedik el, és az egyik okozta félárnyék rávetül a másik objektum okozta árnyékra.

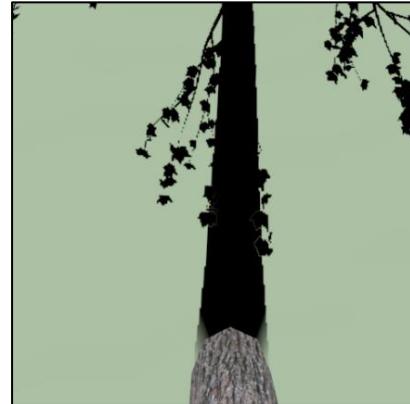
Ennek oka, hogy a Csebisev-egyenlőtlenség nem garantálja a pontos közelítést. A probléma akkor merül fel amikor a szórásnégyzet viszonylag nagy, vagyis a mélységrétekek között nagyobb ugrások vannak. Ebben az esetben a P_{\max} értéke nem közelít olyan gyorsan a 0-hoz, mint mikor a szórásnégyzet kicsi. Ennek következménye, hogy a P_{\max} , a fényesség alsó határa magasabb maradhat.

A fényátszűrődést nem lehet teljesen eltüntetni, de csökkenhető. A megoldás abból indul ki, hogy a teljesen árnyékolt területeken a helytelen félárnyék sosem éri el a teljes intenzitását. Mikor a pixelünk árnyékban van ($z > \mu$), akkor $(z - \mu)^2 > 0$, ami azt eredményezi, hogy $P_{\max} < 1$. Így a helytelenül félárnyéknak számolt területeket

eltávolíthatjuk úgy, hogy bizonyos intenzitás alatt 0-ra állítjuk a $P_{\max} - \sigma$, és a megmaradó értékeket egy lineáris interpolációval átskálázzuk $[0,1]$ közötti tartományba. Ezen kívül az objektumok aljánál keletkező fény-szivárgást csökkenthetjük azáltal, hogy az alacsony szórásnégyzetet rögzítjük egy nála nagyobb, de kis értéknél. [15]



4.9. ábra Fényátszűrődés elmosatlan variancia árnyéktérkép esetében



4.10. ábra Csökkentett fényátszűrődés

4.4.2.2 Shadow acne

A variancia árnyéktérképezés egy hatékony megoldást nyújt az árnyéktolódás problémájára az M_2 momentum felhasználásával. A megoldás [15] lehetővé teszi, hogy az x és y irányú mélységváltozások követésével egy rugalmasabb reprezentációt adjunk a mélységek terjedelmének lefedésére, ezzel csökkentve a shadow acne problémát.

A megoldásban a mélység értékeit egy síkeloszlásként kezeljük, amelyet a következő függvény ír le:

$$f(x, y) = \mu + x \frac{\partial f}{\partial x} + y \frac{\partial f}{\partial y}$$

A mélységeket μ jelöli, a mélységfüggvény gradiensei pedig kifejezik, hogy mennyire változik a mélység x és y irányban. A függvény behelyettesítésével és a pixel szimmetrikus Gauss-eloszlásként való ábrázolásával, megkapjuk az új M_2 momentumot:

$$M_2 = E(f^2) = E(\mu^2) + E(x^2) \left[\frac{\partial f}{\partial x} \right]^2 + E(y^2) \left[\frac{\partial f}{\partial y} \right]^2 = \mu^2 + \frac{1}{4} \left(\left[\frac{\partial f}{\partial x} \right]^2 + \left[\frac{\partial f}{\partial y} \right]^2 \right)$$

Ebben az esetben a szórás fél pixel. A magasabb szórás során jobb eredményeket kapunk, ugyanakkor a fényátszűrődés erősebben jelenik meg. Az árnyéktérkép generálásához használt fragment-shaderben M_2 kiszámítása így alakul:

```

float dx = dFdx(depth);
float dy = dFdy(depth);

Moments.y = depth * depth + 0.25*(dx*dx + dy*dy);

```

4.4.3 Implementáció

A variancia árnyéktérkép erőforrásaihoz egy új osztályt hoztam létre: **VarianceShadowMap**, ami a *ShadowMap* osztállyból származik. A framebuffer-ben lévő képet egy kétcsatornás szín formátummal (R32G32) definiáltam, amiben majd a mélység és mélység négyzet értékek lesznek tárolva. A framebuffer-hez kapcsolt renderpass-ban egyetlen color attachment-et definiáltam.

A variancia árnyéktérkép renderelésénél az egyszerűség kedvéért, a már elkészített szabványos árnyéktérképet használom fel újra és az értékeket módosítom a VSM algoritmus alapján. Ehhez szükséges a textúrát az egész képernyőn megjeleníteni, miközben pontos textúra koordinátákat alkalmazunk. Vulkan-ban a legegyszerűbb mód [5] ennek implementálására, hogy a csúcspontokat nem bufferekben továbbítjuk, hanem a vertex-shaderben számoljuk ki, a `gl_Vertex_Index` bemeneti változóval.

A fragment-shaderben lévő transzformálás során a sampler2D-ként feltöltött árnyéktérképet mintavételezem, majd az árnyéktolódás algoritmus elvégzése után a textúrába írom a mélység és a mélység négyzet értékét.

Ezt követően elmosom az árnyéktérképet a Gauss-elmosással, 5-szemes szűrőt alkalmazva. A megfelelő eredmények érdekében a Pascal-háromszög 12. sorát használtam a súlyok kiszámítására úgy, hogy a középső együtthatókat az összegükkel osztottam.

Az elmosás megvalósítása érdekében egy újabb *renderSystem* osztályt definiáltam **Blur_RenderSystem** néven, aminek az egyetlen feladata a paraméterként kapott árnyéktérkép elmosása. Az elmosás két renderpassban történik. Először egy ideiglenes framebufferbe történik a rajzolás, ahol a képen egy x-irányú elmosást hajtok végre, majd ezt a képet y-irányban is elmosom és a paraméterként kapott framebufferbe rajzolom.

Az elmosott árnyéktérkép mintavételezéséhez létrehozott samplerben beállítottam a lineáris szűrést és engedélyeztem az anizotrop-szűrést is a grafikus kártyám legmagasabb kapacitásán.

A színtér kirajzolásánál a fragment-shaderben implementáltam a Csebisev-egyenlőtlenséget, amivel meghatároztam az árnyék valószínűségének felső határát (P_{\max}). A light-bleeding elkerülése érdekében rögzítettem a szórásnégyzetet egy kis értéknél majd egy lineáris interpolációval átskáláztam a P_{\max} -ot [0.3, 1.0] intervallumban, és a pixel színével szorozva megkaptam a várt eredményt.

4.5 Exponential Shadow Mapping

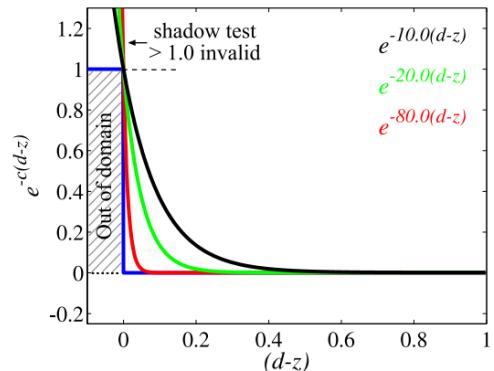
Az exponenciális árnyéktérképezés egy másik egyszerű módszer az árnyéktérkép-szűrés problémájára, ahol az árnyéktesztet egy exponenciális függvénytel közelítjük. Az előzőleg bemutatott VSM technikához képest kevesebb memóriát igényel, illetve kevesebb problémával is jár.

4.5.1 Algoritmus

Az Exponential Shadow Mapping (ESM) [16] algoritmus exponenciális közelítést használ az árnyékteszt során.

Feltéve, hogy $d \geq z$, így definiált az árnyékteszt:

$$f(d, z) = \lim_{\alpha \rightarrow \infty} e^{-\alpha(d-z)}$$



4.11. ábra Árnyékteszt tartomány [16]

A függvény közelíthető egy c nagy pozitív állandóval az α helyett. Ha a c nem elég magas fény-szivárgás problémák léphetnek fel, de egy magasabb érték jobb közelítést nyújt az árnyékteszthez.

$$f(d, z) \approx e^{-c(d-z)} = e^{-cd} \cdot e^{cz}$$

Az árnyékteszt két részre bontható d -től és z -től függően. Így az algoritmus első felét, az árnyéktérkép renderelésekor hajtjuk végre, aztán a tényleges árnyéktesztnél kombináljuk az algoritmus másik felével.

Ez lehetővé teszi, hogy függvény előszűrésével javítsunk az árnyék szélek megjelenítésén már az árnyéktérkép generálásakor. Ha a szűrést konvolúciós operátorként ábrázoljuk és behelyettesítjük az exponenciális közelítést látható, hogy a teljes függvény előszűrése ekvivalens a szűrő közvetlen alkalmazásával az exponens mélységrétekekre.

$$s_f(x) = [w * f(d(x), z)](p) = [w * (e^{-c d(x)} e^{cz})](p) = e^{-c d(x)} [w * e^{cz}](p)$$

4.5.2 Mellékhatások

Az árnyékteszt definiálásánál kimondtuk, hogy a d mélységnek nagyobbnak kell lennie, mint a z mélység. Ebből az adódik, hogy $d(x) - z(p) \geq 0$, csak ez a gyakorlatban nem minden teljesül [16]. Amikor $d(x) - z(p)$ értéke negatív, az árnyéktesztünk nem fog 1.0-hez konvergálni, hanem exponenciálisan fog nőni az eredménye.

A probléma egyik megoldása, hogy mintavételeznél rögzítjük az exponenciális értéket 1.0-nél, és egy PCF-stílusú módszerrel kezeljük az ilyen eseteket. Ezen felül létezik két jobb megoldás: a Z-max Kategorizáció és a Küszöbérték Kategorizáció. Ezt a két módszert, a szakdolgozat keretében nem vizsgáltam, mivel sok esetben még ha egy adott képpontot érvénytelennek is soroltam be, az effektus alig volt látható.

4.5.3 Implementáció

Az ESM algoritmus implementálása nagyban hasonlít ahoz, ahogy a VSM-et készítettem. Ugyanúgy egy külön *renderSystem* végzi a rajzolást, aminek a shaderjében a szabványos árnyéktérkép transzformálása történik. A fragment-shaderben az árnyékteszt z -től, a szabványos árnyéktérképből kiolvasott értéktől függő része szerepel. Az árnyéktérképbe kiírt érték az aktuális pixelből és környezetéből számolt exponenciális értékek átlaga, a c konstanst pedig 80.0 értékkel vettem fel.

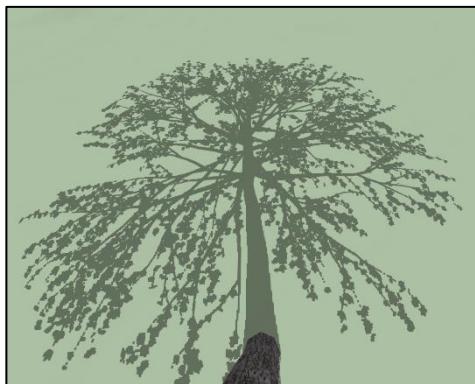
Az árnyéktérkép elkészítése után az 5x5-ös Gauss-szűrőt alkalmaztam a textúrán.

A színtér kirajzolásához az árnyéktérképet feltöltöttem a Main_RenderSystem fragment-shaderjébe, ahol kiszámítottam az árnyékteszt d -től függő részét és szoroztam azt az árnyéktérképből mintavételezett értékkel. Az eredmény esetleges túlcordulása miatt korlátoztam az értékét a [0.3, 1.0] intervallumban. A 0.3 értékkel természetesebb, realisztikusabb hatást érünk el, mintha teljesen feketék lennének az árnyékok.

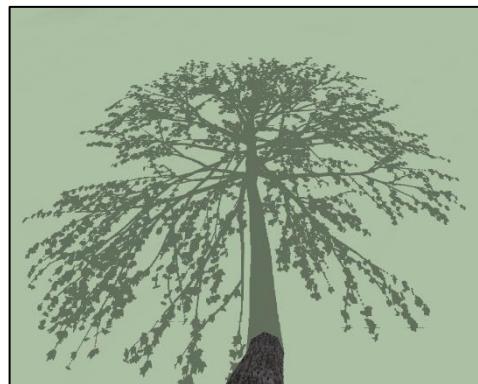
5 Eredmények

A következő fejezetekben bemutatom az implementált árnyéktérkép technikák variációinak eredményeit és azok teljesítmény igényeit.

5.1 Kombinációs lehetőségek



5.1. ábra Simple Shadow Map



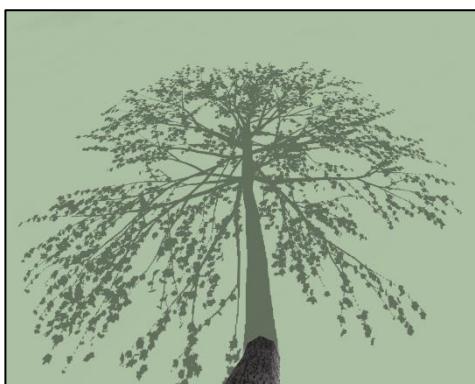
5.2. ábra Cascaded Shadow Map



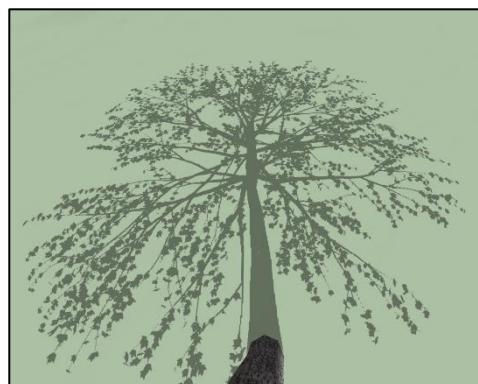
5.3. ábra 4x4 PCF



5.4. ábra 4x4 PCF with CSM



5.5. ábra VSM



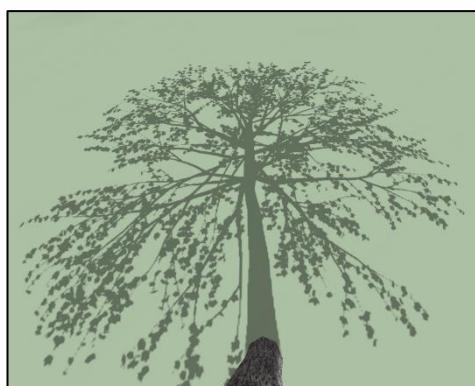
5.6. ábra VCSTM



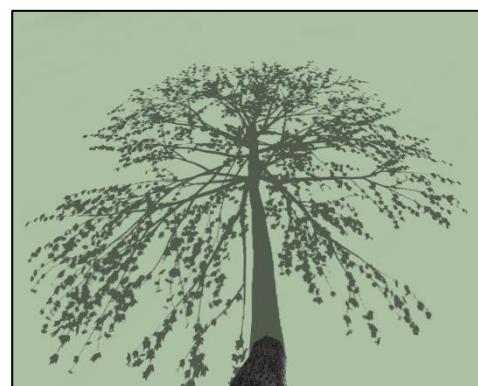
5.7. ábra blured VSM



5.8. ábra blured VCSTM



5.9. ábra pre-filtered ESM



5.10. ábra pre-filtered ECSM



5.11. ábra blured ESM



5.12. ábra blured ECSM

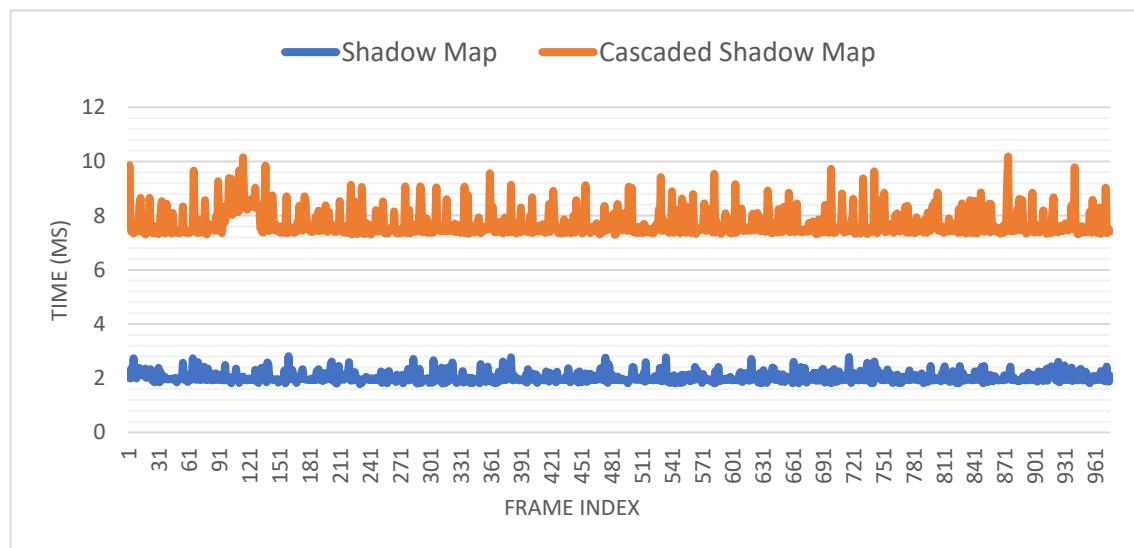
A fenti képeken az implementált árnyéktérképezési technikák eredményei láthatóak. A bal oldali oszlop az általános árnyéktérképezés különböző variációit, míg a jobb oldali oszlop a kaszkádolt árnyéktérképezés változatait demonstrálja. Az árnyéktérképek mérete megegyezik, így láthatóak a különbségek. A valósághűbb eredmények magasabb költséggel járnak, amit a következő fejezetben mutatok be.

5.2 Teljesítménymérés

A teljesítmény méréséhez a Vulkan beépített időbélyeg lekérdezéseit (timestamp queries) használtam. Timestamp query-k segítségével az egyes árnyéktérképek renderelési idejét mértem milliszekundumban.

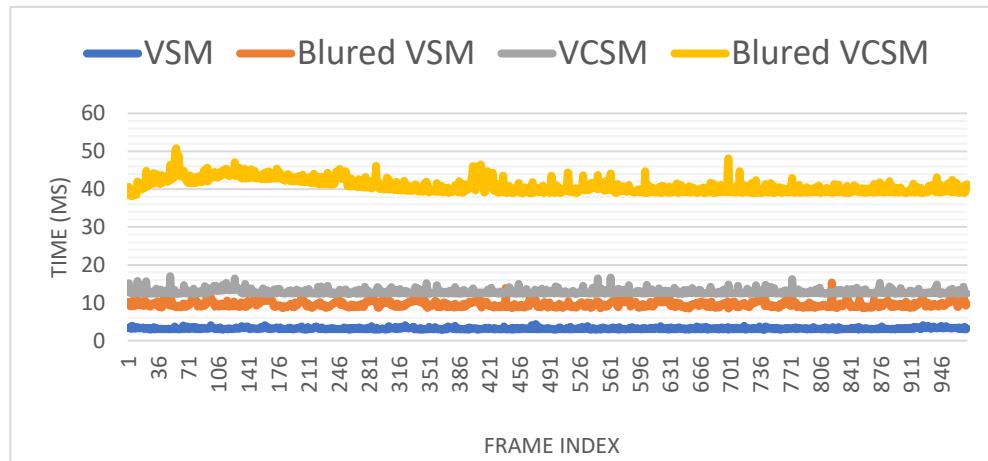
A mérésben szereplő árnyéktérképek 2048x2048 méretűek. A kirajzolt terrain 135x135 csúcspontból áll, a modell pedig közelítőleg 1.3 millió csúcspontból. A felhasználói ablak mérete 800x600 képpont volt.

A következő összehasonlító diagram az általános és a kaszkádolt árnyéktérkép generálásának idejét mutatja be. A kaszkádolt árnyéktérképezés több erőforrást igényel és képkockasebessége is alacsonyabb, viszont néhány esetben a módszer komplexitás és minőség aránya jobb, mint az általános módszeré. Jelen mérésben a kaszkádolt árnyéktérkép renderelési ideje körülbelül háromszor hosszabb az általános módszerhez képest. Ennek oka, hogy a látóteret 4 darab kaszkádra osztottam, vagyis a kaszkádolt algoritmus elkészítéséhez 4 darab árnyéktérképre volt szükség.



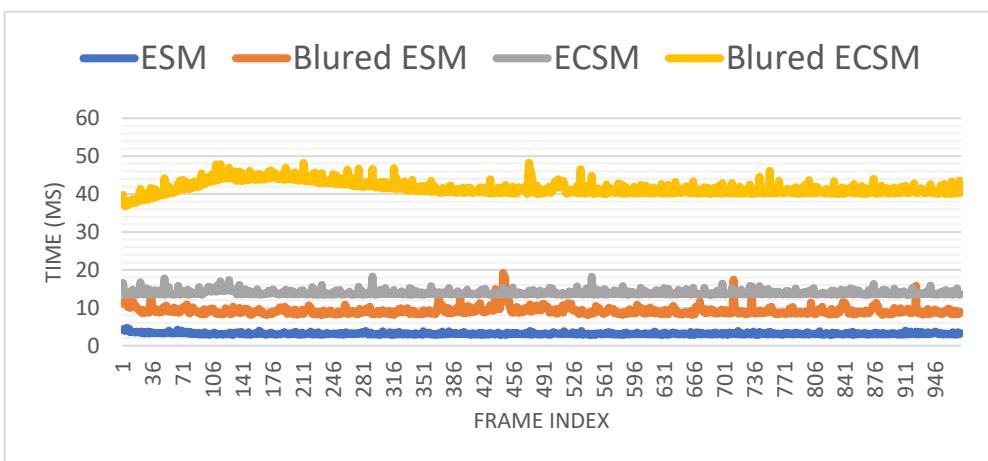
5.13. ábra Az általános árnyéktérkép és a kaszkádolt árnyéktérkép rajzolási idejének összehasonlítása GPU timestamp-ek alapján

A variance shadow mapping (VSM) algoritmus kombinációs lehetőségeinek összehasonlítását a következő diagram mutatja. A variancia árnyéktérkép létrehozásához a már előzőleg elkészített általános árnyéktérképet mintavételezem és transzformálom. Ennek a mintavételezésnek és transzformációnak az átlagos időigénye 1000 frame-re számolva átlagosan 1.19 milliszekundum, ami ebben az esetben hozzáadódik az általános árnyéktérkép renderelési idejéhez. Az elmosott variancia árnyéktérkép létrehozásához a megalkotott variancia árnyéktérképet használom, amin egy 5x5-ös Gauss-szűrést végzek.



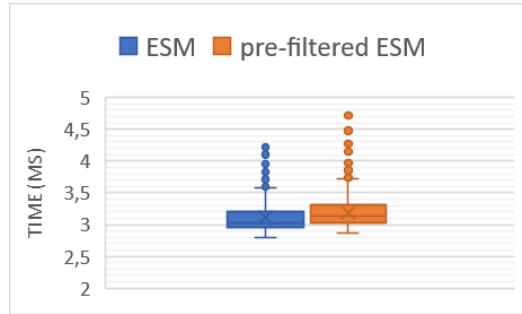
5.14. ábra A VSM variációk elkészítési ideje

Az ESM verzióinak elkészítése hasonló módon történik, ezért az eredmények közel azonosak. Ez azért lehetséges, mert az exponenciális árnyéktérképet is kétsatornás színtextúrával hoztam létre. A GPU optimalizálása érdekében a tovább fejlesztési lehetőség, hogy az exponenciális értékek tárolása egy egycsatornás árnyéktérképben történjen.



5.15. ábra Az ESM variációk elkészítési ideje

Az ESM algoritmusból adódóan az árnyéktérképet már a renderelésnél előszűrhetjük. Jelen esetben egy 2x2-es szűrést használtam. Az árnyéktérképek létrehozásának időbeni eltérése minimális, viszont a megjelenésben jól látható javulást érhetünk el. A következő diagram ábrázolja a létrehozási időben mért eltérést, majd a képek szemléltetik a minőség javulását:



5.16. ábra Az exponenciális árnyéktérkép létrehozási idejének összehasonlítása az előszűrt változattal



5.17. ábra Exponenciális árnyéktérképezés



5.18. ábra Előszűrt exponenciális árnyéktérképezés

6 Összegzés

Mindig is foglalkoztatott a videójátékokban szereplő virtuális valóságok mögötti algoritmusok működése, amibe a szakdolgozat elkészítése által betekintést nyerhettem.

A feladat elkészítése során megismerkedtem a grafikai programozás különböző területeivel és a Vulkan API-val. Mélyebben megértettem az árnyékolás szerepét és működését a virtuális világok létrehozásában, ennek hatására sikerült jobban átlátnom, hogy mennyire bonyolult a valóság pontos ábrázolása a képernyőn.

Az elkészített dolgozatban vizsgáltam az árnyék megjelenítési technikákat, azon belül is az árnyéktérkép módszereket és változataik egymáshoz illeszthetőségét. Ezáltal azt a következtetést tudom levonni, hogy a realisztikus árnyékok implementálása valós idejű, dinamikusan változó környezetben egy nagyon összetett és kihívásokkal teli probléma.

Irodalomjegyzék

- [1] Vulkan® 1.1.270 - A Specification (with all registered Vulkan extensions), <https://registry.khronos.org/vulkan/specs/1.1-extensions/html/vkspec.html>
- [2] Alexander Overvoorde, Vulkan Tutorial: <https://vulkan-tutorial.com/>
- [3] Joey de Vries, Learn OpenGL: <https://learnopengl.com/>
- [4] Blinn. Simulation of Wrinkled Surfaces, Siggraph 1978
- [5] Sascha Willems, "Vulkan tutorial on rendering a fullscreen quad without buffers": <https://www.saschawillems.de/blog/2016/08/13/vulkan-tutorial-on-rendering-a-fullscreen-quad-without-buffers/>
- [6] Efficient Gaussian blur with linear sampling:
<https://www.rastergrid.com/blog/2010/09/efficient-gaussian-blur-with-linear-sampling/>
- [7] Williams, L. 1978. Casting curved shadows on curved surfaces. In Proc. SIGGRAPH, vol. 12, 270–274.
- [8] F. C. Crow. 1977 Shadow algorithms for computer graphics. Computer Graphics (Proceedings of SIGGRAPH '77), pages 242–248
- [9] Shirley, Peter (July 9, 2003). Realistic Ray Tracing. A K Peters/CRC Press; 2nd edition. ISBN 978-1568814612.
- [10] Reeves, W. T., D. H. Salesin, and P. L. Cook. 1987. "Rendering Antialiased Shadows with Depth Maps." *Computer Graphics* 21(4) (Proceedings of SIGGRAPH 87).
- [11] Dimitrov, R. (2007). Cascaded shadow maps. Developer Documentation, NVIDIA Corp.
- [12] Zhang Fan, Hanqui Sun, Oskari Nyman. "Parallel-Split Shadow Maps on Programmable GPUs", nvidia developer, GPU Gems 3. Chapter 10.
- [13] Johan Medestrom. 2016. "OpenGL Cascaded Shadow Maps": <https://johanmedestrom.wordpress.com/2016/03/18/opengl-cascaded-shadow-maps/>
- [14] Donnelly, William, and Andrew Lauritzen. 2006. "Variance Shadow Maps." In *Proceedings of the Symposium on Interactive 3D Graphics and Games 2006*, pp. 161–165.
- [15] Andrew Lauritzen. "Summed-Area Variance Shadow Maps", nvidia developer, GPU Gems 3, Chapter 8.

- [16] Annen, T., Mertens, T., Seidel, H. P., Flerackers, E., & Kautz, J. (2008). Exponential shadow maps. In Graphics Interface (pp. 155-161). ACM Press.
- [17] GLFW: Multi-platform library for OpenGL, <https://www.glfw.org/>
- [18] tinyobjloader: <https://github.com/tinyobjloader/tinyobjloader>
- [19] GLM: <https://github.com/g-truc/glm>
- [20] STB: <https://github.com/nothings/stb>
- [21] Dear ImGui : <https://github.com/ocornut/imgui>
- [22] Wave .obj file: [Wavefront .obj file - Wikipedia](#)