Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Irányítástechnika és Informatika Tanszék

Fridvalszky András Máté

TÖBBMINTÁS ÉLSIMÍTÁS ALKALMAZÁSA DEFERRED SHADING ESETÉBEN

TDK dolgozat

KONZULENS

Dr. Tóth Balázs Görgy

BUDAPEST, 2019

Tartalomjegyzék

Összefoglaló **4**
Abstract **5**
Köszönetnyilvánítás **6**
1 Bevezetés **7**
1.1 Deferred shading 7
1.2 Többmintás élsimítás (MSAA) 9
1.3 Deferred shading többmintás élsimítással 11
2 Kapcsolódó eredmények **13**
2.1 Deferred shading és MSAA 13
2.2 Variable rate shading (VRS) 15
3 A javasolt algoritmus alapjai **16**
4 A javasolt algoritmus **18**
4.1 Index map alapú kapcsolat 18
4.2 Fordított címzés 20
4.3 Láncolt lista alapú kapcsolat 21
5 Tesztelt algoritmusok és a környezet bemutatása **23**
5.1 Közös jellemzők 23
5.1.1 Keretalkalmazás 23
5.1.2 Általános beállítások 24
5.1.3 Fényforrások kezelése 25
5.1.4 Árnyalás 25
5.1.5 Áttetsző objektumok 26
5.1.6 Árnyékok 26
5.1.7 Hagyományos G-Buffer felépítése 27
5.2 Tesztelt algoritmusok 27
5.2.1 Deferred (élsimítás nélkül) 27
5.2.2 FXAA 28
5.2.3 MSAA 28
5.2.4 MSAA Z-prepass használatával 29
5.2.5 A javasolt algoritmus 30
6 A mérési folyamat **32**

6.1 Színterek 32
6.1.1 1. Színtér 32
6.1.2 2. Színtér 33
6.1.3 3. Színtér 34
7 Eredmények összehasonlítása 35
7.1 Memóriahasználat 35
7.2 Futás idő 37
8 Összefoglalás 44
9 További lehetőségek 45
9.1 Memória foglalás feldarabolása 45
9.2 Változó mintaszám 46
9.3 Z-Prepass megszüntetése 46
9.4 Platform bővítés 47
Irodalomjegyzék 48

Összefoglaló

A deferred shading alapú grafikus eljárások népszerűek a valós idejű háromdimenziós alkalmazások körében, mivel nagyságrendekkel több fényforrás használatát teszik lehetővé, mint a hagyományos forward shading alapú módszerek. Ennek a megközelítésnek azonban hátránya, hogy a GPU által hardveresen támogatott többmintás élsimítás (MSAA) alkalmazását nem támogatják automatikusan. Erre a problémára több megoldás is létezik, de közös negatívumuk, hogy nagy mértékben növelik a megjelenítő memória használatát, illetve sávszélesség igényét. Emiatt manapság inkább az útófeldolgozás alapú élsimító eljárásokat részesítik előnyben (pl.: FXAA).

Ezek a technikák, ahelyett, hogy magasabb frekvenciával mintavételeznék a képet, megkeresik és elsimítják az éleket. Ez a módszer jellegéből adódóan sokkal gyorsabb, de nem tud minden esetben tökéletes eredményt nyújtani és könnyen életlen képet, vagy például gyors kamera mozgásnál zavaró hibákat eredményezhet.

Dolgozatomban áttekintem a deferred shading működését, különböző változatait, előnyeit és hátrányait, illetve az eddig ismert eljárásokat a több mintás élsimítás alkalmazására. Ezután olyan új eljárást mutatok be, amely segítségével a szokásos módszerekhez képest sikeresen lehet csökkenteni a memória és sávszélesség igényt úgy, hogy a képminőség a többmintás élsimításhoz képest nem változik lényegesen.

Az új módszert eltérő méretű és karakterisztikájú színtereken mutatom be, és összehasonlítom a széles körben elterjedt eljárásokkal, mind teljesítmény, mind memóriaigény szempontjából, egy Vulkan alapú megjelenítő segítségével.

Abstract

Deferred shading based rendering algorithms are popular with real time three dimensional applications, because they make possible of using orders of magnitude more light sources than with classical forward shading algorithms. The disadvantage is, that we cannot use the built-in multisample anti-aliasing algorithms of the GPU (MSAA). There are multiple solutions for this problem, but the increased memory and bandwidth consumption of the renderer is a common drawback. For this reason, nowadays it is typical to use post processing based anti-aliasing methods (e.g.: FXAA).

These techniques try to find and then blur edges on the picture, instead of sampling it with higher frequency. The inherent consequences of these methods are that they are much faster than MSAA but they cannot always produce correct results, the picture could become blurry or fast camera movement could result in visible artifacts.

In my essay I make an overview of different variations of deferred shading algorithms, their advantages and disadvantages and the known techniques for using multisample anti-aliasing with them. Then I will propose a new method that can successfully decrease memory and bandwidth requirements of classical solutions while maintaining quality of the picture.

I am going to present it using a Vulkan based renderer on scenes with different sizes and characteristics while comparing performance and memory usage to the current techniques.

Köszönetnyilvánítás

Köszönettel tartozom konzulensemnek, Dr. Tóth Balázsnak, aki mindig örömmel adott tanácsot és segített munkámban.

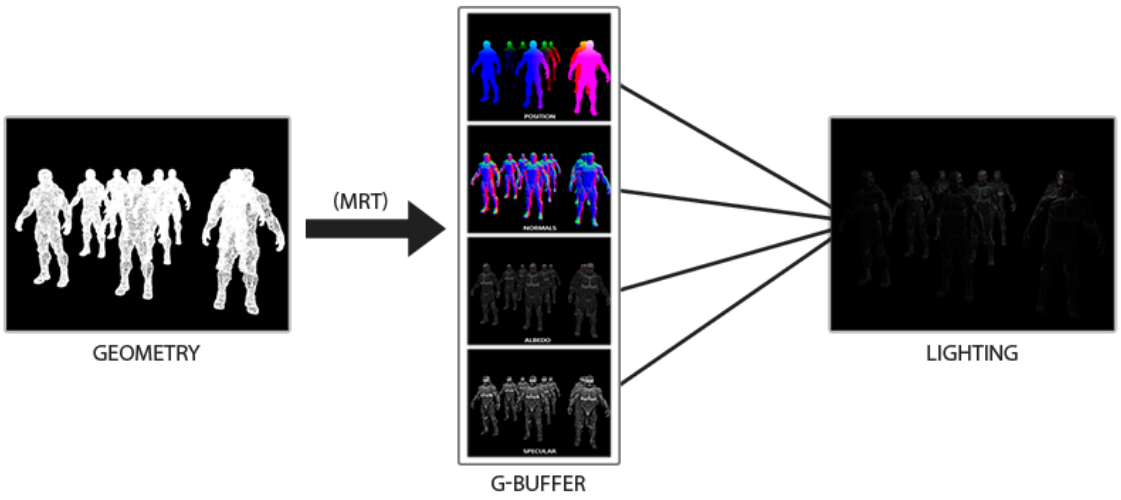
Az egyes színterekhez felhasznált modellek Morgan McGuire archívumából származnak. (https://casual-effects.com/data)

1 Bevezetés

Egy háromdimenziós megjelenítő implementálása sok nehéz problémát vet fel, amelyek esetében a döntés a különböző megoldások között nem egyértelmű. Továbbá ezek a megoldások gyakran kihatnak a teljes rendszerre és összefüggenek egymással is. Emiatt vannak olyan algoritmusok amelyek jobban, illetve olyanok amelyek kevésbé férnek meg egymás mellett. Amennyiben az utóbbiak közül szeretnénk egyszerre többet alkalmazni, akkor sokszor kompromisszumokat kell kötnünk melyek a teljesítmény vagy minőség rovására mehetnek. Ilyen algoritmus pár az alapvető megjelenítési struktúrát meghatározó Deferred shading és a többmintás élsimítás (Multisample anti-aliasing, MSAA). Az alábbiakban ezt a két módszert és a köztük felmerülő problémát mutatom be.

1.1 Deferred shading

Ezt a technikát először 1988-ban vetették fel, de mind a mai napig elterjedt a számítógépes játékok körében. A célja az, hogy csökkentse, illetve lehetőség szerint megszüntesse a feleslegesen elvégzett munkát az egyes képpontok fényforrások szerint való árnyalásakor. Ezek a felesleges számítások a kitakart képpontok színezéséből adódnak, amelyek nagyon sok erőforrást tudnak igényelni, például fotorealisztikus árnyalás esetén. A problémát deferred shading algoritmus úgy oldja meg, hogy a tényleges árnyalás előtt összegyűjti egy ideiglenes tárolóba (G-Buffer) a majdani színezéshez szükséges adatokat (diffúz szín, normál stb.), minden látható pixelre. Ezt hívják geometry pass-nak. A következő lépésben pedig végrehajtja az árnyalást, amely így, hogy rendelkezésre áll a teljes geometriai információ akár fényforrásonként párhuzamosítható is. Erre lighting pass néven szoktak hivatkozni. Ezzel a megközelítéssel a képelőállítás bonyolultsága a geometriából adódó képpontok és a fényforrások számának szorzata helyett a látható képpontok és a fényforrások számának összegére csökken.



1.1. ábra: A deferred shading működése. Az MRT a multiple render target-nek a rövidítése. Ez OpenGL esetében az egyszerre több textúrába történő rajzolást jelentette. (forrás: learnopengl.com)

A deferred shadingnek is több változata van, ezek általában a lighting pass implementációjában térnek el. A legismertebb, amire általában csak simán deferred shading (vagy rendering) néven szoktak hivatkozni, úgy működik, hogy a fényforrásokhoz hozzá rendel egy geometriát, ami a fényforrás "hatóterét" reprezentálja. Ez pontfényforrás esetén egy gömb, aminek a mérete az erősségétől függ, irányfényforrás esetén pedig egy teljes képernyőt lefedő téglalap. A lighting pass során ezek a geometriák kerülnek kirajzolásra. A GPU elvégzi a raszterizálás során azoknak a pixeleknek a kiválasztását, melyek a fényforrás hatni fog. A fragment shaderben megtörténik az adott pixelre az árnyalás és az eredményeket egy közös textúrába kell összegezni. Ezen utána el lehet végezni a szükséges utómunkálatokat.

A tiled deferred shading nevű változat úgy működik, hogy felbontja a kamera frustumát két (vagy akár három) dimenzióban kisebb darabokra. Egy compute shader a mélység buffer alapján kiválogatja, hogy melyik fényforrás melyik darabokba esik bele és ezeket egy láncolt listába felfűzi. Utána az árnyaláshoz szintén compute shader-t lehet használni, ami minden pixel esetében a megfelelő láncolt listában tárolt fényforrásokon iterál végig.

A deferred lighting nevű változat három lépésből áll. Itt a geometry pass során egy kis méretű G-Buffer készül el, amiben csak a fény összegyűjtéséhez szükséges információk vannak benne (mélység, normál). Ezután következik a fényforrásokból beérkező fény összegyűjtése, hasonlóan a deferred shading-hez, a G-Bufferben lévő adatok alapján. Az utolsó lépés az árnyalás, amihez a teljes geometriát újra ki kell rajzolni,

de mivel a teljes beérkező fény minden képpontra ki lett számolva, az árnyalás egyből elvégezhető.

1.2 Többmintás élsimítás (MSAA)

A háromszögek raszterizálásakor diszkrét pontokban mintavételezzük a geometriai információkat. A korlátos mintavételi frekvencia miatt a geometriai primitívek határainál, ahol ugrásszerűen változik az adott mintavételi pontban látható felület fellép az aliasing hatás. Ez a gyakorlatban a recés, lépcsős élekben figyelhető meg melyek zavaróak és elrontják az összképet. A kamera mozgása ezt a jelenséget fel is erősítheti, még látványosabbá téve ezeket a hibákat. Ennek a problémának a megoldására alkalmazhatóak az úgynevezett élsimító eljárások (anti-aliasing) amelyek valamilyen módon lokálisan megnövelik a mintavételi frekvenciát a problémás képpontokban.

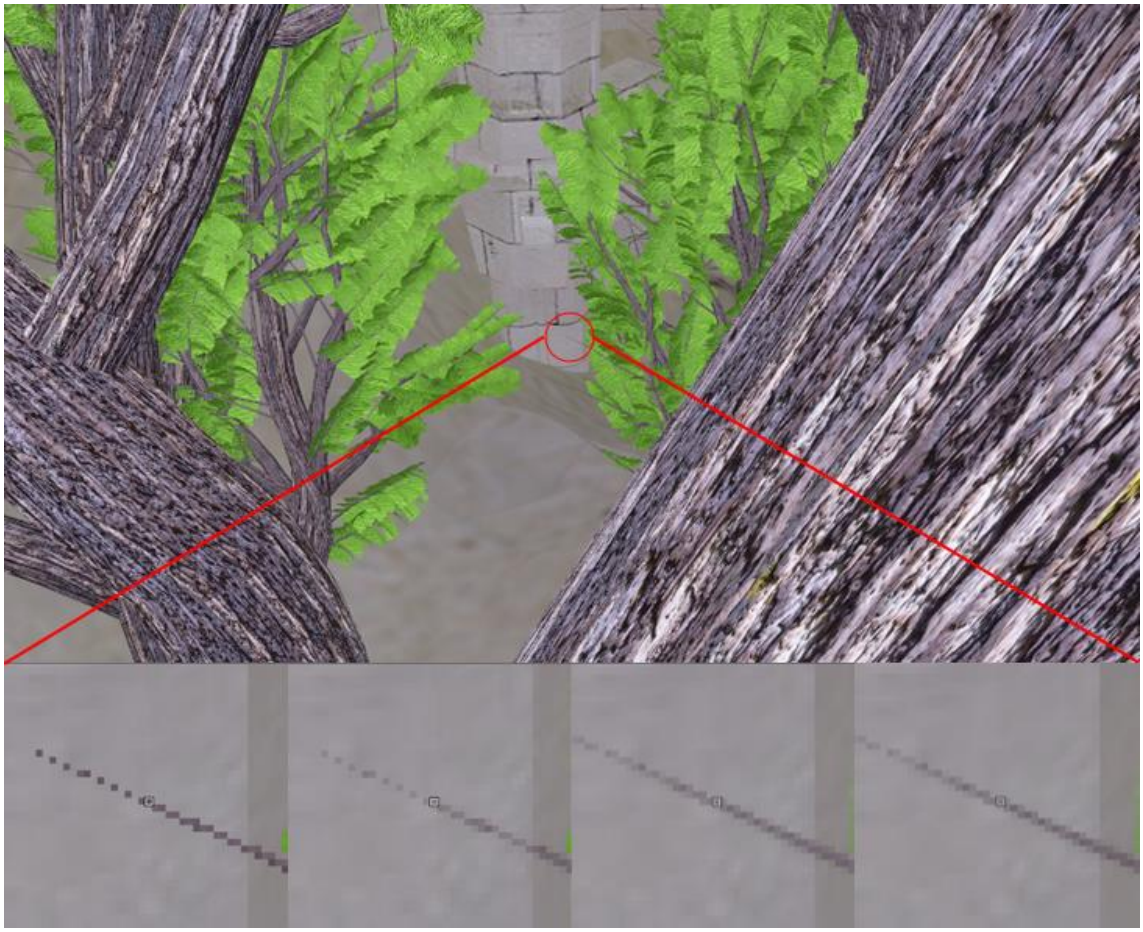
Ezek közül a legegyszerűbb megoldás a kép magasabb felbontásban történő mintavételezése (kirajzolása), majd megfelelő szűrő segítségével történő újra mintavételezése (leskálázása) a cél felbontásra (supersampling). Ez a művelet a probléma gyökerét veszi célba (az alacsony mintavételezési frekvenciát) így az összes vizuális hibát megoldja, viszont nagyon nagy az erőforrás igénye ezért gyakorlatban csak ritkán alkalmazzák valós idejű alkalmazások esetén.

Ennek egy optimalizált változata a multisample anti-aliasing, amely hasonlóan magasabb felbontáson mintavételezi a képet, de ezt csak ott próbálja megtenni, ahol ténylegesen szükség van rá. Ez az algoritmus a GPU-ban általában hardveresen támogatott. A működése a következő:

Minden képponthoz több értéket, úgynevezett mintát tárol el, amelyekhez eltérő képponton belüli pozíció tartozik. Amikor a raszterizáció során egy háromszög metsz egy képpontot, akkor a GPU azt is megvizsgálja, hogy melyik mintákat metszette el. Ezután az árnyalást csak egyszer hajtja végre úgy mintha az a pixel közepén történne, de a kapott eredményt az összes metszett mintához tartozó memóriaterületre kiírja. Ez azt jelenti, hogy minden háromszög-képpont metszés csak egy színezés végrehajtását fogja jelenteni. Ha a teljes kép előállt, akkor az utolsó lépés az egy képponthoz kiírt minták átlagolása és az eredmény kiírása.

Ez általában azt jelenti, hogy egy képpont esetében akkor fogjuk többször végrehajtani az árnyaló algoritmust, ha több háromszög is metszi. Ez két objektum

határán is előfordul, ahol általában megjelenik az aliasing így ezeken a helyeken megoldja a problémát, elsimítja a zavaró éleket. Hátránya viszont, hogy ez előfordul egy objektumon belül is, a modellt alkotó olyan háromszögek határán, ahol közel folytonos a felület és itt feleslegesen végez többletmunkát. Ez a többletmunka a látható háromszögek méretétől függően lényegesen kevesebb lehet, mint amennyit a teljes kép túlmintavételezése jelentett volna. Szintén probléma, hogy csak a geometriai határokkal foglalkozik így az egyéb okokból (például megvilágításból, árnyékok határaiból) adódó aliasingot nem tudja orvosolni.



1.2. ábra: Az élsimítás hatása. Az alsó képek balról jobbra haladva: nincs élsimítás, FXAA, 8x MSAA, az új algoritmus (8x)

A fenti ábrán látható, hogy az FXAA nem képes rekonstruálni az apró ág eredeti alakját. Megmarad a darabos struktúra, pusztán kevésbé lesz látható az elmosás hatására. Ezzel szemben az MSAA és a dolgozatomban bemutatott új algoritmus képes megtartani az ág tömörségét a magasabb frekvencián történő mintavételezés segítségével.

1.3 Deferred shading többmintás élsimítással

A két technika kombinációja nem lehetetlen feladat, de több problémát is felvet és sok hátránya van. Mindkét módszer önmagában is magas memória használattal és sávszélesség igénnyel rendelkezik, amely együtt már érezhetően a teljesítmény rovására megy. Ahhoz, hogy alkalmazhassuk az MSAA-t a teljes G-Buffer-t többszörös mintavételezéssel kell eltárolnunk. Ez az élsimításhoz szükséges minták számával arányosan növeli a méretét. A szokásosan választott 2, 4, illetve 8 minta esetén a G-Buffer memória igénye is 2, 4, illetve 8-szorosa lesz. A mai grafikus kártyák esetében ez a méret növekedés elfogadható, de mobil eszközöknek általában nincsen ekkora grafikus memóriája. További probléma viszont, hogy írni és később olvasni is kell ezt a memóriát, ami sávszélesség problémákhoz vezethet, akár asztali gépek esetében is.

Eltekintve a memória problémáktól egyéb gondok is felmerülnek. Miután elkészült a G-Buffer végre kell hajtani minden mintára az árnyalást, majd képpontonként átlagolni az eredményeket. Természetesen, ha ezt így tesszük meg, akkor vissza jutunk a supersampling-hez, amit épp elkerülni próbálunk. Erre a problémára megoldást jelent, ha a G-Buffer-ben eltároljuk a lefedettség információt is, ami azt adja meg, hogy egy adott képpontban elhelyezkedő mintákat hány különböző háromszög metszette. Ha ezt csak egyetlen háromszög tette meg, akkor elég egyszer végrehajtani az árnyalást. Ennek a döntésnek az implementálása sem egyértelmű, mivel dinamikus elágazást hozhat be a shader kód végrehajtásába, ami teljesítmény vesztést jelenthet. A stencil buffer használatával megoldható enélkül is, de ez egyéb plusz költségeket fog okozni.

Régebben az is probléma volt, hogy csak modern hardverben voltak meg azok az eszközök melyek segítségével egyáltalán implementálni lehetett a felvázolt megoldást. Manapság ez már nem gond, de fenti hátrányok továbbra is jelen vannak. Ezért nem is szokás együtt alkalmazni a két módszert, hanem helyette gyorsabb és a deferred shadinggel kompatibilisebb élsimítási technikákat alkalmaznak. Ilyen például az FXAA, MLAA, SMAA [1][2][3] is.

Ezek az algoritmusok nem magasabb frekvenciával történő mintavételezést használnak, hanem a rendes felbontású képen végeznek élkeresést majd a megtalált éleket összemossák. Így tehát nem a probléma gyökerét célozzák meg, hanem csak a tünetet próbálják kezelni több-kevesebb sikerrel. Ebből kifolyólag, ha nem elég agresszív az algoritmus, akkor recés élek maradhatnak a képen, ellenkező esetben viszont életlenné

válhat. Az eredeti színek is módosulhatnak és minimálisan, de torzíthatják a képet. A gyors mozgás is problémás lehet, mivel az élkeresés sokszor bináris döntést hoz, ami apró, de zavaró ugráláshoz vezethet a képen.

Dolgozatomban a deferred shading és az MSAA együtt történő alkalmazásának vizsgálom egy olyan új megoldását, amely lecsökkenti a memória használatot és a sávszélesség igényt, jobb teljesítményhez és nagyobb rugalmassághoz vezetve a klasszikus módszerekhez képest. Az új technika alapja, hogy nem tárolja feleslegesen az ismételt mintákat, megszünteti a redundáns információt és ezzel javítja a teljesítményt. Mivel a szükséges minták száma képről képre változik a javasolt új módszer adaptívan alkalmazkodik ehhez.

2 Kapcsolódó eredmények

Korábban is léteztek megoldások a deferred shading és az MSAA egyszerre történő alkalmazására, amelyeket az alábbiakban foglalok össze. Az itt alkalmazott ötletek részben a később bemutatott algoritmusban is felhasználásra kerültek. Emellett itt tárgyalom a Variable rate shading-et is, amely hasonló probléma megoldására alkalmas.

2.1 Deferred shading és MSAA

Az Nvidia publikált egy példát [4], melyben DirectX 11-et használva bemutatták, hogy hogyan lehetséges a két technika ötvözése. A mélység textúrát és a G-Buffer-t több mintás textúraként hozták létre. Ezekbe írt a geometry pass egyszerű MSAA-t használva. Ezután több lehetséges megoldást is adtak a folytatásra.

A következő lépésben már nem állt rendelkezésre automatikusan az eredeti geometriai információ, amelyből kiderül, hogy melyik képpontokat elég csak egyszer árnyalni és hol (illetve milyen mértékben) szükséges több minta kiértékelése. Ezért ezt az információt szükségképpen hozzá kellett venni a G-Buffer-hez. A kérdés az volt, hogy ezeket a képpontokat, hogy lehet megtalálni. Egyszerű képpontoknak nevezték azokat, amelyeket csak egy egyedi minta alkotott, összetettnek az összes többit. Ezek szétválasztására a cikkben két megoldást is adtak.

Az első az egyszerű MSAA során is használt lefedettség vizsgálata volt. Ez megadta, hogy a shader hívás mely mintákhoz tartozik az adott képponton belül. Ha az összes mintára érvényes volt, akkor egyszerű, ellenkező esetben összetettként jegyezte fel.

A második megoldás a képponton belül elhelyezkedő minták mélységéből, normáljából és színéből próbálta detektálni, hogy egyszerű vagy összetett-e a pixel.

A második megoldás jobb eredményeket nyújtott, nem jelölte meg mindig a háromszögek oldalait, de cserébe lassabb volt a végrehajtása, az összes mintát meg kellett vizsgálni hozzá. Az első megoldás több összetett képpontot eredményezett, de a detektálás nem igényelt további erőforrásokat a tároláshoz használt memóriaterületen kívül.

Az árnyalást és átlagolást egy lépésben hajtották végre, amelyre több némileg eltérő megközelítést is javasoltak.

Az első esetben a fragment shaderben kiolvasták, hogy az előző lépés milyen döntést, hozott, egyszerű-e vagy összetett-e a képpont. Ha egyszerű volt, akkor egyszer árnyaltak egy mintát és annak az eredményét adták vissza. Ha összetett volt, akkor pedig supersampling-et alkalmaztak, azaz minden mintát árnyaltak és a kapott eredmények átlaga került a kimenetre.

A második módszer abban tért el az előzőtől, hogy a döntést nem a shaderben hozták meg. Létrehoztak egy stencil buffert, ami különválasztotta az egyszerű és összetett képpontokat. Ezután két lépésben árnyaltak, először az egyszerűeket, utána pedig az összetetteket.

Az első esetben a hátrányt a dinamikus elágazás jelentette. A GPU, akkor a leghatékonyabb, ha minden shader lefutása azonos utasításokat követ végig, csak az értékek változnak. Ennek oka, hogy a GPU a shaderek lefutását úgynevezett warp-okba rendezi. Ha egy warp-on belül az egyik shader több időt vesz igénybe mint a többi, mert eltérő lefutási utat választott, akkor a többinek is várnia kell. Emiatt ez az elágazás akár jelentős teljesítmény csökkenést is jelenthetett.

A második esetben ezt kerülték el azzal, hogy direkt megadták a GPU-nak melyik képpont hogyan fog viselkedni. Ehhez azonban a stencil buffert is el kellett készíteni, illetve egy helyett két rajzolási utasítást kellett kiadni.

Egy harmadik lehetőséget is felvetett a cikk a felesleges árnyalás számítások csökkentésére. Ehhez nem csak azt kell eltárolni, hogy komplex vagy összetett-e a képpont, hanem a teljes (mintánkénti) lefedettség információt is. Ezt felhasználva a fragment shaderben lehetőség nyílik arra, hogy pontosan meg lehessen határozni melyik mintákat kell árnyalni és azokat milyen súllyal kell figyelembe venni az átlagoláskor.

Amikor teszteltem ezt az implementációt, akkor az első probléma az volt, hogy a GPU a lefedettséget mindig a mélység (és stencil) teszt előtt határozza meg. Ez viszont azt is jelenti, hogy azokban az esetekben, amikor egy képpontban két háromszög metszi egymást, akkor előfordulhat, hogy külön-külön mind a két háromszögben belső pontokat találunk. Ebben az esetben a tesztelés eredményeként is azt kapjuk, hogy az adott képpont egyszerű, holott mind a két háromszög hozzájárulását figyelembe kellene vennünk. Erre

a megoldást a *post-depth coverage* kiegészítés jelenti, amely pont ezt a problémát oldja meg.

2.2 Variable rate shading (VRS)

A 2018-ban megjelent Turing architektúrájú GPU-kal vezette be az Nvidia ezt az új technológiát [5], amely szoros kapcsolatban van dolgozatom témájával. A korábbi GPU-k egyáltalán nem, vagy csak nagyon korlátozottan (kiegészítéseken keresztül) támogatták a különböző számú mintával rendelkező textúrákba történő rajzolást. A Variable rate shading ezt teszi sokkal rugalmasabbá.

A programozó átadhat a GPU-nak egy speciális textúrát, ami minden 16x16 képpontot tartalmazó területhez a képernyőn eltérő rátájú árnyalást rendelhet. Ez jelentheti azt, hogy minden mintára külön fragment shader hívás jut (supersampling) vagy akár azt is, hogy egy 4x4-es területre csak egy darab. Ez a technológia főleg virtuális valóságot megjelenítő alkalmazásokat célozza meg, amely területen a konzisztensen magas megjelenített képkocka szám az elvárás, míg csak a képernyő bizonyos részein van szükség éles képre. A kép szélein vagy a program természetéből adódó (például gyors mozgás esetén elmosódó) területeken nincs. Itt a VRS egy jó megoldás lehet a teljesítmény koncentrálására a képernyő fontos területeire. Így akár ugyanazzal teljesítmény igénnyel lehet elérni részletesebb és szebb megjelenítést, vagy a hardverrel szemben támasztott követelmények csökkenését.

A későbbiekben bemutatott módszer nem csak az MSAA-t és a deferred shadinget köti össze, hanem a supersampling lokális szabályozását is lehetővé teszi a VRS-hez hasonlóan. Továbbá elég rugalmas ahhoz, hogy a régebbi GPU-kon is elérhetővé tegye közvetlen hardveres támogatás nélkül a VRS többi funkcionalitását, némileg kisebb teljesítmény nyereséggel.

3 A javasolt algoritmus alapjai

Az alábbiakban az algoritmus működésének alapötletét mutatom be. A felmerülő problémák részletezést egyelőre kihagyom, ezek felvetésére és megoldására a későbbi részletesen bemutatott változatok során fogok visszatérni.

A hagyományos G-Buffer általában néhány, az alkalmazás felbontásával megegyező méretű textúrából áll. A textúrák egy-egy texelje tárolja a hozzá tartozó képpontban (illetve mintában) kiszámolt geometriai információkat. Általában ezeket az adatokat a méret minimalizálás céljából tömören, akár speciális kódolással tárolják.

A bemutatott módszer első lépése ennek a struktúrának a megszüntetése. Erre azért van szükség mivel a textúrákhoz inherensen hozzá tartozik, hogy minden mintához előre lefoglaljuk az összes szükséges memóriát melynek nagy része általában kihasználatlan marad. Ehelyett egy memória foglalás alapú módszert vezet be, melyhez hasonlót a sorrend független átlátszóság, láncolt lista alapú implementációjában szoktak alkalmazni. A G-Buffer így egy tömbbé és egy számlálóvá válik. A tömb elemei lesznek az egy-egy mintához tartozó adat blokkok, nem definiált sorrendben. A számláló kezdetben a tömb elejére fog mutatni. A geometry pass során minden shader hívás mely érzékeli, hogy új memóriaterületre van szüksége, egy atomi művelet segítségével megnöveli a számlálót lefoglalva az aktuálisan mutatott blokkot. Így a tényleges memória foglalás és írás minden háromszög-képpont metszés esetén csak egyszer történik meg. Az összesen lefoglalt memória mérete képkockáról képkockára változhat, de általában a maximum jóval alacsonyabb lesz mintha textúrákat használtunk volna.

Az így létrejött struktúra tömören és redundancia nélkül tárolja az árnyaláshoz szükséges információt. Hiányzik viszont az összeköttetés, hogy az egyes adat blokkok, mely képponthoz tartoznak (textúra esetén ez implicit módon adódott a koordinátákból). Ennek a problémának a különböző lehetséges megoldásaira a későbbiekben részletezett implementációk során térek majd ki.

Fontos megemlíteni néhány azonnal látható előnyét ennek a megoldásnak a hagyományos textúra alapú G-Buffer-ekhez képest:

- Az összetartozó, azaz egy időben írt-olvasott adatok memória szinten folytonosan vannak tárolva így a hardver hatékonyabban tudja kiolvasni

az adatokat, mintha különálló textúrákból kellene összeszednie őket. A nyereség implementáció függő, de semmiképp sem negatív.

- Az előbbi pont miatt, a hagyományos esetben, érdemes kevesebb textúrát alkalmazni, azonban azokat a lehető legteljesebben kihasználni, hogy a memória hozzáférés hatékonyabb legyen. Viszont a textúra méretek nem tetszőlegesen választhatók, hanem csak olyanokat lehet használni, amiket a GPU megenged. Így gyakori eset, hogy választani kell a hatékonyabb elérés és a feleslegesen lefoglalt memóriaterület között. Ezzel szemben a tömb alapú megoldást nem kötik a textúra, illetve annak írásának és olvasásának a sajátosságai. Be kell tartani az alignmenttel[1] kapcsolatos szabályokat, de az megtehető, hogy két vagy több blokkot a tömb egy elemeként kezeljünk. Ezzel úgy lehet megszüntetni felesleges mezőket, hogy a lokalitás csak kicsit romlik (néhány blokknyi eltolás). Ennek a működése pontosan látszani fog egy példán keresztül a későbbiekben (a konkrét implementáció leírásakor).

A következő lépés az árnyalás és a minták manuális átlagolása. Ehhez minden képponthoz azonosítani kell az őt alkotó mintákat, majd egyenként árnyalni, végül pedig átlagolni kell. Ennek a technikája közvetlen attól függ, hogy milyen módon lett eltárolva a fentebb vázolt kapcsolat a képpontok és adatblokkok között.

[1] A GPU a legtöbb CPU-hoz hasonlóan hozzárendel egy úgynevezett alignmentet minden primitív típushoz. Ez azt jelenti, hogy az adott típusú változó csak olyan memóriacímen kezdődhet, ami ennek többszöröse. Ha két eltérő alignmenttel rendelkező változó helyezkedik el egymás után, akkor szükséges lehet, hogy a fordító paddinget (nem használt byteokat) illesszen a kettő közé, hogy a második változó is helyes memóriacímen kezdődhessen.

4 A javasolt algoritmus

Az alábbiakban részletesen bemutatom a fejlesztés és tesztelés során megvizsgált különböző módszereket a fentebb vázolt problémák megoldására. Az egyes módszerek általam tapasztalt előnyeit és hátrányait is tárgyalom egyelőre teljesítmény mérések nélkül.

4.1 Index map alapú kapcsolat

A G-Buffer a fentiek szerint épül fel, azaz egyetlen tömbből áll és a számláló kezdetben a tömb elejére mutat. A memóriaterület lefoglalása is a fentiek szerint történik. A memória lefoglalásakor felmerülő és megoldandó első probléma az overdraw, amely azt jelenti, hogy az előrébb lévő fragmens felülírja a hátrébb lévő korábban sorra került fragmenset. Ez azért válik különösen súlyossá ebben az esetben, mert itt nem csak teljesítmény vesztést, hanem „memória szivárgást” is okoz. Ez elrontja az adatok redundancia mentességét és aláássa a módszer alapvető célját.

A korábban lefoglalt és később haszontalanná vált memória blokkok dinamikus összegyűjtése, vagy akár csak a szivárgás detektálása még bonyolultabb struktúrákat és többlet számolást igényelne. Emiatt egyelőre az egyszerűség kedvéért az első lépés egy Z-prepass, amely meghatározza az egyes képpontokban a legközelebbi felületi pont távolságát. Ezt követi a geometry pass, ahol igénybe veszem a stencil buffert is. Erre azért van szükség, mivel Z-prepass-t követően lehetne olyan esetek, ahol két metsző háromszög raszterizálása ugyanazt a mélység értéket eredményezi egy pontban. A stencil buffer segítségével meg lehet számolni az egy mintára eső fragmenseket és azok közül csak az elsőt átengedni.

A képpontokkal való kapcsolat megteremtésének módja ebben a verzióban a lehető legegyszerűbb lesz. Fel kell venni a tömb mellé egy új textúrát, az index map-et. Ez egy több mintás textúra lesz, amely minden mintához egy számot tárol, ami a tömbben a hozzá tartozó adat blokkra mutat. A textúra a geometry passban lesz feltöltve, a lefoglalt blokk indexét kell az aktuális mintához kiírni. Mivel egy memória blokkhoz (azaz háromszög-képpont metszéshez) csak egy shader lefutás tartozik, az általa kiírt érték a metszéshez tartozó összes mintához kiíródik.

Az árnyalás és átlagolás egy lépésben történik. Egy teljes képernyőt lefedő négyszöget kell kirajzolni, mélység buffer nélkül egy minta/képpont beállítás mellett. A fragment shaderben ki kell olvasni az adott képponthoz tartozó indexeket minden mintára egy ciklusban. Ez alapján már meg lehet szerezni az adatokat a G-Buffer-ből, végre lehet hajtani az árnyalást és az eredményeket lehet átlagolni. Felmerül a kérdés, hogy lehet-e, illetve van-e értelme kezelni az ismétlődő indexeket. Én az első implementációban eltároltam a már árnyalt indexeket, és ismétlődés esetén felhasználtam az elmentett eredményeket, ahelyett, hogy újra számoltam volna őket. A korábbi deferred shading, MSAA kombinációkban ez a kérdés fel sem merült, mivel ott nem állt rendelkezésre egy azonosító (az index) a mintához. Ennek megfelelően, vagy csak egy mintát számoltak ki, vagy pedig supersampling-et alkalmaztak, a lefedettség alapján.

Ennek a rendszernek a hátrányai a következők:

- A Z-prepass használata magas vertex számú színterek esetén drasztikusan csökkentheti a teljesítményt.
- Az index map jelenléte, melyre a sima deferred shading esetén nincsen szükség. Ez növeli a memória használatot (amit épp csökkenteni szeretnénk) és sávszélesség igényt.
- A minták átlagolása dinamikus elágazásokat tartalmaz, ami nem ideális a teljesítmény szempontjából.
- A memória foglalás is egy szűk keresztmetszetet jelent, mivel minden shader hívás ugyanazt a számlálót változtatja atomi utasításokkal, ezeket valahol a hardverben sorosítani kell.

A G-Buffer ketté választásával optimalizálhatjuk a memória hozzáférést. Az első rész, amely mérete megegyezik a számított képpontok számával, különleges lesz abból a szempontból, hogy mindig fog ide kerülni érvényes adat. Ehhez szükséges, hogy minden képpontra minden képkockában rajzoljunk valamit, de ez sokszor (például játékok esetében) amúgy is teljesül. A geometry pass során a fragment shaderben meg kell vizsgálni az aktuális lefedettséget. Ha egy előre kijelölt indexű minta (például a 0-ás) le van fedve, akkor új memóriaterület foglalása helyett használhatjuk a speciális (“előre lefoglalt”) területet a G-Buffer elején.

Az egyik előnye ennek a módosításnak, hogy nagy mértékben csökkent a terhelés a globális számlálón. A rendszer is rugalmasabbá vált, mivel nem lesz probléma, ha kevés memória lett lefoglalva. Ha ilyen helyzet áll fenn, akkor is lesz legalább egy minta minden képponthoz. Ezt az árnyaláskor le lehet kezelni és az eredmény egy rugalmas módszer lesz, mely a rendelkezésre álló memória függvényében a képernyő több-kevesebb részén végez élsimítást.

A hátránya csupán az, hogy egy dinamikus elágazás került be a geometry pass-ba.

4.2 Fordított címzés

Az árnyalásra és átlagolásra egy másik lehetséges megközelítés, amit kipróbáltam a következő. A tömbben tömören szerepel az összes minta, amit a későbbiek során árnyalni kell. Ezt hatékonyan, elágazások nélkül meg lehet tenni egy compute shader segítségével is. Ez azt jelenti viszont, hogy elveszítjük a fények geometriai információit melyet az árnyalás párhuzamosításához és a nem megvilágított képpontok kihagyásához használtunk. Ezért ebben az esetben a deferred shading-nek csak a tiled deferred shading nevezetű változatát lehet használni, amelyet korábban tárgyaltam.

További probléma lesz viszont az átlagolás, mivel a compute shader a fenti rendszerben egyszerre csak egy adat blokkot lát, így ehhez egy külön lépésre lesz szükség. Egyik lehetőség, hogy az árnyalás eredményét egy másik több mintás textúrába írom ki. Ehhez szükség van az adat blokkhoz tartozó koordinátát és lefedettséget is eltárolni a G-Buffer-ben. Viszont mivel megfordult a kapcsolat iránya ezért nincs már szükség az index map-re. Az eredmény textúra átlagolását ezután már közvetlen a GPU segítségével lehet elvégezni, amely a Vulkan API használata esetén egyetlen függvény hívást jelent.

A fő hátránya ennek a megoldásnak az, hogy az egyes lépéseket agresszívan sorosítani kell. Nem kezdődhet el az árnyalás, amíg a teljes tömb el nem készült. Nem kezdődhet el az átlagolás, amíg az árnyalás be nem fejeződött.

Előnye az előző rendszerhez képest, hogy a dinamikus elágazások fragment shaderből compute shaderbe kerültek át, illetve nem a drága árnyalás közben vannak jelen, hanem csak az eredmény kiírásakor melyet a GPU sokkal könnyebben tud kezelni. Kisebb lesz a lehetséges várakozás a párhuzamosan futó shader hívások között is.

4.3 Láncolt lista alapú kapcsolat

A következő módszer az előző kettőhöz hasonlóan szintén a kapcsolatok leírásának módját változtatja meg. Az index map helyett a G-Buffer minden adat blokkjához még egy érték kerül felvételre, amely egy mutató. A cél az, hogy a mutatók segítségével minden képponthoz egy-egy láncolt lista alakuljon ki a hozzá tartozó mintákat reprezentáló blokkokból a geometry pass végére.

A láncolt listák fejét az első néhány blokk tárolja, pontosan annyi ahány képpont van. Ezek a blokkok úgy fognak működni, mint a G-Buffer kettéválasztását alkalmazó optimalizálás során tárgyaltak. Ha az aktuális shader hívás során memóriaterületet kellett foglalni, akkor ennek az új blokknak az indexét egy atomi művelet segítségével be kell kötni a képponthoz tartozó láncolt listába. Ez úgy működik, hogy az új blokk indexe kerül a láncolt lista elejére, a korábbi kezdő tagra pedig az új elem fog mutatni.

Így már rendelkezésre áll egy képponthoz az azt alkotó minták, de hiányzik a lefedettség. Valójában nem is a teljes lefedettségre van szükségünk, hiszen az egyes minták egyenrangúak. Elég csupán az, hogy egy adat blokk hány mintát reprezentál. Kihasználva, hogy tudjuk hány mintának kell összesen lennie, elég csupán minden dinamikusan lefoglalt blokkhoz eltárolni ezt a számot,és az előre lefoglalt blokkhoz tartozó érték ebből már visszakapható. Mivel a mutató 32 bites, de ebből nincs szükség az összes bitre, a felső néhány (8 minta esetén 3) bitet felhasználhatjuk erre a célra. Ez az érték fogja megadni, hogy a mutatott adat blokk hány mintát reprezentál.

Az árnyalás és átlagolás ezek után újra egy lépésben elvégezhető. Egy shader hívás fog kezelni egy képpontot. Kiszámolja a képponthoz tartozó kezdő blokk árnyalását, majd az esetlegesen jelenlévő láncolt listát is bejárja és megszínezi a mintákat. A kapott értékeket súlyozza és átlagolja az alapján, hogy egy blokk hány mintát reprezentál.

Fontos előnye ennek a módszernek az előzőekhez képest, hogy csökkent a G-Buffer mérete. Nincs szükség minden lehetséges mintához eltárolni egy mutatót (index map), hanem csak annyit amennyi adat blokkot a rendszer felhasználhat. Az adatok továbbra is redundancia mentesek így akár a compute shader hatékony használatára is lehetőség nyílik, ha egyéb szempontok miatt (pl.: tiled deferred shading) ez szükséges lenne. De a rendszer továbbra is működik a hagyományos deferred shading-el is.

Hátrány továbbra is, hogy az átlagolás során a ciklus iteráció száma dinamikus, amely lassíthatja azoknak a képpontoknak a számítását is, ahol csak egy minta van. Ezt a korábbiakhoz hasonlóan a tiled deferred shading tudná javítani.

Az előzetes tesztek alapján az utoljára bemutatott algoritmus nagyságrendekkel jobban teljesített a korábbiaknál, mind memória használat, mind teljesítmény szempontjából. Emiatt a későbbi részletes összehasonlítások során csak ennek a módszernek az eredményeit írom le.

5 Tesztelt algoritmusok és a környezet bemutatása

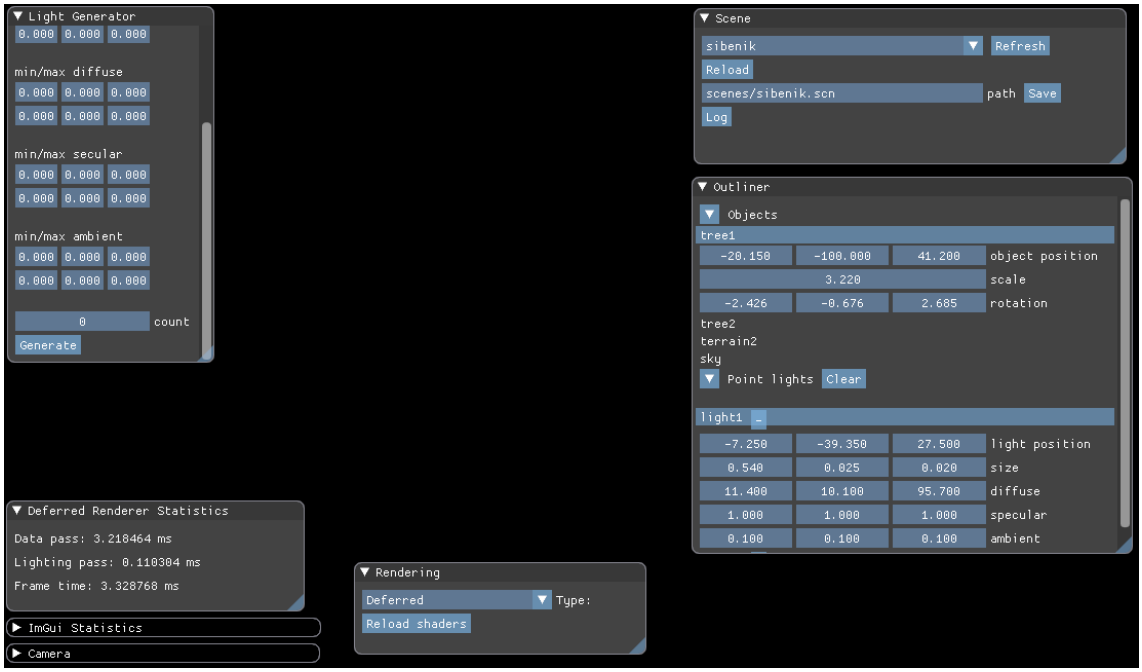
Az alábbiakban először a tesztelt módszerek közös jellemzőit tárgyalom. Ezt követően kitérek az egyes módszerek pontos jellemzőire, eltéréseire és arra, hogy mi célt szolgálnak az összehasonlítás során.

5.1 Közös jellemzők

A Deferred shading implementálásakor több különböző módszer közül is lehet választani. A dolgozatomnak nem célja ezeknek az összehasonlítása, ezért az összes tesztelt algoritmus számára egyetlen közös megoldást választottam, hogy az eredmények összehasonlíthatóak legyenek. Ugyanezen okból kifolyólag minden esetben azonos árnyalási technikát alkalmaztam. Az algoritmusokat egy közös Vulkan alapú keretrendszer segítségével implementáltam.

5.1.1 Keretalkalmazás

Az algoritmusokat Vulkan és C++ segítségével implementáltam egy közös alkalmazásban. A célja ennek a rendszernek az volt, hogy gyorsítsam a fejlesztés menetét és elősegítsem a majdani tesztelést. A felhasználói felület a következőképpen nézett ki:



5.1. ábra: Az alkalmazás felhasználói felülete.

A rendszer az alábbi funkciókat támogatta futásidőben:

- megjelenítők közötti váltás
- megjelenítők paramétereinek állítása
- shader kódok újratöltése
- teljesítmény monitorozása, fájlba mentése
- színterek közötti váltás, mentés
- fényforrások paramétereinek állítása
- fényforrások törlése és véletlenszerű generálása
- objektumok transzformációjának módosítása
- szabad kamera mozgás
- rugalmas, dinamikusan átméretezhető UI (ImGui)

A megjelenítők kód szinten egymástól függetlenek voltak. Egy közös rendszert használtak, ami elérhetővé tette számukra a Vulkan használatát és elvégezte a megjelenítéshez feltétlen szükséges erőforrások kezelését.

A Vulkan-ra azért esett a választásom, mivel jó rálátást és hozzáférést enged a GPU-hoz. Segítségével pontosabban megtudtam határozni, hogy épüljön fel a rendszer és biztos lehettem benne, hogy ezt az eszköz nem bírálja felül. Több Vulkan nyújtotta lehetőséget is kihasználtam a fejlesztés során, ezek közül az egyik leghasznosabb a subpass-ok voltak. Ezek lehetővé teszik a GPU számára, hogy egy képkocka elkészítésének menetét az egyes lépések közötti függőségek figyelembevételével optimalizálja és csökkentse a memóriahasználatot. Ezt, ahol lehetőség volt rá használtam is az egyes módszerek megvalósításakor.

5.1.2 Általános beállítások

A korai mélység tesztek minden technika esetén bekapcsolásra kerültek. Emellett, ahol értelme volt (pl. MSAA) az úgynevezett *post-depth coverage* kiegészítést is alkalmazták a megjelenítők. Ez tette lehetővé, hogy a lefedettség értékek a tényleges mélység és stencil teszt utáni eredményt tükrözzék.

Mindegyik megjelenítő használt egy 24+8 bites mélység (és stencil) buffert. A 24 bites mélység felbontás elégséges volt az alkalmazáshoz, a stencil buffert pedig több

algoritmus is használta. Mindegyik textúra lineáris színtérben került beolvasására. Ahol a textúra sRGB színtérbeli nemlineáris információt tárolt (pl.: albedo) ott ennek linearizálása kézzel történt az érték kiolvasását követően. A kimeneti textúra sRGB típusú volt, így nem volt szükség manuális gamma korrekcióra.

5.1.3 Fényforrások kezelése

Az eltérő deferred shading implementációk fontos eleme a fényforrások kezelésének módja. Erre sok lehetőség kínálkozik, különböző következményekkel a teljesítményre nézve. Céljuk a felesleges árnyalás számítások elkerülése, illetve párhuzamosítása. Ezek a módszerek arra alapoznak, hogy egy fényforrás nem hat az összes képpontra, hanem csak a környezetében lévőkre.

Én ennél egy sokkal egyszerűbb koncepciót választottam. Nálam minden fényforrás minden képpontra hat kivétel nélkül. Ennek az oka az volt, hogy a mérések könnyebben értelmezhetővé váltak. A tesztelt módszerek teljesítmény különbségének fontos alkotóeleme, hogy hányszor kell kiszámolni az árnyalást. A fentebb említett optimalizációkkal ennek a számát lehetne csökkenteni, azonban ez, ugyanahhoz az eredményhez vezetne az összes tesztelt algoritmus esetében. Az implementált változattal viszont közvetlen a minták és fényforrások számából látható, hogy hányszor lett árnyalás végrehajtva.

A fényeket egy globális tömb segítségével (uniform buffer) implementáltam. Ebben egymás után helyezkedtek el a fényforrások adatai melyekre részletesen az árnyalás tárgyalása során térek ki. Egy minta feldolgozásakor a fényforrások hatását egy ciklusban kell sorban kiszámolni és összegezni.

Ez a módszer a fentiek miatt tehát nem az optimális, valós környezetben is alkalmazandó megoldás, ellenben a mérések egymáshoz viszonyított eredményeit nem rontja el.

5.1.4 Árnyalás

Valós alkalmazások esetén az árnyalás számít az egyik leginkább teljesítményigényes résznek. Ez határozza meg azt is, hogy milyen információkra lesz szükség a G-Buffer-ben. Emiatt, ha a mérésekhez egy olyan, például egy fotorealisztikus játék esetén nem feltételezhető algoritmust, választok (pl.: phong-árnyalás), amihez kisebb G-Buffer is elégséges, akkor a kapott eredmények nem fogják tükrözni a deferred

shading és MSAA kombinációjából adódó memória problémák megoldásának jelentőségét.

Mivel a fizikai alapú árnyalások (PBR) elterjedtek a modern játékmotorok közt és kellően nagy mennyiségű adatot igényelnek ezért ideális választást jelentettek számomra. Közülük is a Cook-Torrance [6] árnyalást választottam az elterjedtsége és a hozzá tartozó asset-ek könnyű elérhetősége miatt. Az algoritmust kamera térben hajtottam végre.

A fényforrások paraméterei ennek megfelelően egy ambiens és egy diffúz erősségből, illetve egy konstans lineáris és négyzetes lecsengési együtthatóból álltak.

Az árnyalásból kapott eredményeket HDR adatként kezeltem és fényforrásonként egyszerűen összegeztem. Ez az adat azonban nem alkalmas arra, hogy élsimítást végezzünk rajta. Ennek oka, hogy majd a képernyőn nem ezek az értékek fognak megjelenni, el kell végezni előtte a tone mapping-ot is. Ha ez előtt a lépés előtt hajtják végre az élsimítást, akkor kontrasztos területeken a tone mapping visszahozhatja a recés éleket. Én az egyik egyszerűbb, lokális módszert, a Reinhard tone mapping-ot [7] választottam.

5.1.5 Áttetsző objektumok

Az alkalmazásban nem kezeltem az áttetsző objektumokat. Ez egy probléma hagyományos deferred shading esetén is, és a megoldás általában az, hogy az áttetsző objektumokat külön rajzolják ki, a többi után, forward shading segítségével. Ez az új algoritmus esetében is változtatás nélkül alkalmazható. A dolgozatomban nem vizsgáltam meg olyan módszereket, amelyek a javasolt algoritmussal potenciálisan jobban tudnának együttműködni.

5.1.6 Árnyékok

Árnyékokhoz leggyakrabban a shadow mapping-et (és ennek változatait) használják, melyek deferred shading esetén is problémamentesen alkalmazhatóak. Ez, hasonlóan az áttetsző objektumokhoz, nem változik meg az új algoritmus esetében sem.

5.1.7 Hagyományos G-Buffer felépítése

A G-Buffer a következő adatokat tartalmazta: albedo, képtérbeli normálvektor, metallic, roughness és ambient occlusion (ao) tagok. Ezeket az alábbi módon tároltam 2 texúrában:

1. bit			24.	32. bit
albedo (RGB)				metallic

1. bit	32.	40.	48.	64. bit
normál		roughness	ao	-

5.2. táblázat: A G-Buffer-t alkotó 2 textúra felépítése. A felső ábra egy 4 db 8 bites komponensből álló, az alsó egy 4 db 16 bites komponensből álló textúrára utal.

A normálvektor 3 komponensét 2 tag használatával kódoltam el Lambert Azimuthal Equal-Area Projection [8] segítségével. Egy tag számára 16 bit pontosság jutott. A többi elem tagonként 8-8bitet használt. A második textúra végén egy 16-bites blokk kihasználatlan maradt, a későbbiekben itt kerültek tárolásra az egyes technikákhoz szükséges egyéb adatok. Összesen egy mintához tehát 96 bit (12 byte) memóriára volt szükség.

A színezendő képpont kamera térbeli pozíciója azért nem szerepel a G-Buffer-ben, mivel ez a mélység buffer-ben található értékből kiszámítható, ismerve a képpont képtérbeli koordinátáit.

5.2 Tesztelt algoritmusok

Az alábbiakban az egyes algoritmusok működésében lévő eltéréseket, illetve a fentiekhez képesti esetleges különbségeket emelem ki. Az egyes módszerek választásának okát is tárgyalom a későbbi mérésekben betöltött szerepük alapján.

5.2.1 Deferred (élsimítás nélkül)

A dolgozatomnak nem képezi témáját az élsimítást használó algoritmusok összehasonlítása az élsimítást nem használóval. Ellenben a különböző élsimítások teljesítményének összehasonlításához hasznos támpontot nyújt, ha egy közös alapnak veszem ezt. Azaz úgy tekintek a kérdésre, hogy mi a költsége az egyes élsimító eljárásoknak. Egy ilyen mértékkel az egyes színterek közötti teljesítménykülönbségek könnyebben lesznek értelmezhetőek és összehasonlíthatók.

5.2.1.1 Működése

A geometry pass során a textúrák megfelelő értékei beolvasásra kerülnek a fragment shaderben. Ezeket az értékeket a G-Buffer képpont szerint adott koordinátáira kell kiírni, a normálvektort kamera térbe transzformálva és tömörítve, a többit változatlanul hagyva.

Az árnyalás során ezek az értékek kerülnek vissza olvasásra. Ez minden képpontra egyszer történik meg, ez egy a teljes képernyőt lefedő téglalap kirajzolásával tehető meg. Itt egy ciklusban kell végrehajtani az összes fényforrás alapján a Cook-Torrance árnyalást. Az eredményeket összegezni kell majd tone mapping segítségével LDR adattá alakítani. A kapott értékek kerülnek megjelenítésre.

Ez Vulkan esetében 1 renderpass és azon belül 2 subpass segítségével megvalósítható, mert az árnyalás során mindig csak az aktuális képponthoz kiírt adatokra lesz szükség. Ez a korlátozása a subpassoknak.

5.2.2 FXAA

Hasonlóan az előző ponthoz, nem célja dolgozatomnak ennek az eljárásnak és az MSAA alapú technikáknak az összehasonlítása. Többek között azért is, mivel a döntés a két fajta megoldás között az egyéni vizuális igények alapján is történhet, a teljesítménybeli különbségek mellett. Csupán a teljesség kedvéért szerepel a tesztelt módszerek között, hogy az éldetekció alapú technikák is képviselve legyenek.

5.2.2.1 Működése

Az eljárás majdnem teljesen azonos az élsimítás nélküli Deferred shadinggel. A különbség az, hogy megjelenik egy második renderpass. Ez az előző pass eredményén dolgozik és végrehajtja az FXAA eljárást. Ez röviden éleket keres a képpontok közt luminancia alapján. Ha megtalál egy élet, akkor simítást alkalmaz az azt alkotó képpontokon. A második renderpass szükséges, mivel az FXAA egy képpont környezetét vizsgálja meg.

5.2.3 MSAA

Ez a technika reprezentálja dolgozatomban a manapság használt algoritmusokat a Deferred shading és MSAA kombinációjára. Mivel célom egy olyan új algoritmus

bemutatása mely ezeknek a karakterisztikáin (memória, sebesség) javít, ezért ezt a módszert fogom vele a dolgozatomban összehasonlítani.

5.2.3.1 Működése

A két technika kombinálásához a G-Buffert alkotó textúráknak is multisampled-nek kell lennie. Ezt követően a geometry passt a kívánt MSAA beállítások mellett kell végrehajtani. Emellett fontos változás, hogy a lefedettséget is el kell menteni. Szerencsére ehhez a G-Buffert nem kell bővíteni, az 1 bit információ, hogy a képpont egyszerűnek vagy összetett-e elfér az üresen maradt területen.

Az árnyalást is módosítani kell. Ugyanúgy, ahogy korábban minden képpontra egyszer kell lefutnia a fragment shadernek, tehát 1 db téglalap kirajzolása szükséges. A shader először visszaolvassa a hozzá tartozó képpont összes mintájából azt, hogy egyszerű-e vagy összetett. Ha mindegyik azt állítja, hogy egyszerű akkor a működése a továbbiakban megegyezik az élsimítás nélküli Deferred shadinggel. Ellenkező esetben a korábban már tárgyalt árnyalást minden mintára végre kell hajtani és az eredményeket átlagolni.

Mivel nincs szükség arra, hogy az átlagolás során több képpont adataihoz is hozzáférjünk, ezért a fentieket egy renderpasson belül két egymást követő subpass segítségével lehet hatékonyan implementálni.

5.2.4 MSAA Z-prepass használatával

Egy logikus kritikája lehet a teszteknek, hogy az egyik fő tesztalany használ Z-prepasst, viszont a másik nem. Ez speciális színterek esetén jelenthet olyan teljesítmény változást, amely torzítaná az eredményeket. Emiatt, hogy az ilyen színtereket detektálni lehessen ezt a módszert is bevettem az összehasonlításba. Főleg az alacsony vertex számú, nagyon sok átfedést tartalmazó színterek okozhatnak ilyen problémát.

5.2.4.1 Működése

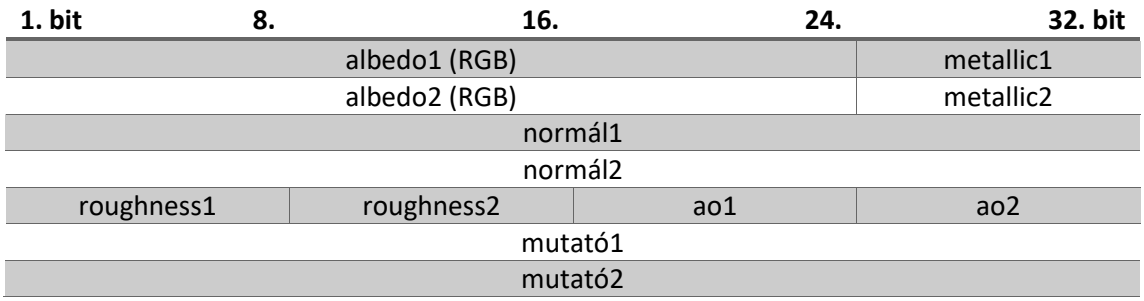
Ebben az esetben annyiban módosul a Deferred MSAA, hogy a geometry pass előtt egy Z-prepass is végrehajtásra kerül. Ezután a geometry passt lehetett futtatni olyan mélység teszttel, ami csak egyenlőség esetén engedi át a fragmenset. Emellett szükséges volt a stencil buffer használata is, hogy az azonos mélységgel rendelkező fragmensek közül csak egy juthasson tovább.

5.2.5 A javasolt algoritmus

A dolgozatom célja ennek az új technikának a bemutatása és összehasonlítása a manapság ismert és használatos módszerekkel. Ez az algoritmus rendelkezett a legjobb karakterisztikákkal a dolgozatomban korábban tárgyalt lehetséges módszerek és implementált optimalizációk között, ezért ezt választottam ki az összehasonlításhoz.

5.2.5.1 Működése

Az algoritmus első lépése egy Z-prepass futtatása. Ezt követi a G-Buffer feltöltése a geometry pass során. Itt is használatra kerül a stencil buffer, hogy minden mintához biztosan csak egyszer fusson le a fragment shader. A G-Buffer egy blokkjának a felépítése:



5.3. ábra: A G-Buffert alkotó tömb egy elemének felépítése. Egy elem 2 minta adatait tartalmazta.

Egy mintához 112 bit (14 byte) hosszú adat blokk tartozik. Ebben szerepel egy 32 bites mutató, amelyen atomi műveleteket szeretnénk a későbbiekben végrehajtani. Emiatt, ha a G-Buffert alkotó tömb egy eleme ilyen hosszú lenne, akkor alignment miatt 16 bites paddinget kellene minden elem végére beilleszteni. Ezt elkerülendő a tényleges tömb egy eleme két ilyen mezőt fog tartalmazni, azaz 224 bit hosszú lesz. Ez picit ront az adat lokalitáson, de a szükséges memória méretet is csökkenti.

A mutató helyére beírt érték az alábbi elemekből épül fel:

5.4. ábra: A mutató felépítése. A minták száma maximum 8 lehetett ezért elfért 3 biten. Az 1 minta indexére a multisampled mélység buferből való olvasás miatt volt szükség. Ez is maximum 8 különböző értéket vehetett fel.

Maga a G-Buffer a következőképpen néz ki:

5.5. ábra: A G-Buffer felépítése. Az előre lefoglalt blokkok száma a megjelenítendő pixelek számával egyezett meg.

A következő szabad elemet jelző mutatót is be kell állítani méghozzá úgy, hogy az előre lefoglalt elemek után mutasson eggyel, azaz a megjelenítendő képpontok számával legyen egyenlő. A dinamikusan lefoglalt blokkok számának megválasztása az elérhető memóriától, teljesítménytől és kívánt eredménytől függ.

A geometry pass során a fragment shader-ben az első lépés annak az eldöntése, hogy szükséges-e új memóriaterületet foglalni. Ehhez elég a lefedettséget jelző bitmaszkot megvizsgálni, hogy a 0. minta benne van-e. Ha igen, akkor használhatjuk az előre lefoglalt területből az aktuális képpont koordinátáinak megfelelő mezőt. Ha nem, akkor egy atomi művelettel meg kell növelni a globális mutatót. A visszakapott érték lesz a lefoglalt memóriaterület indexe. Mielőtt ezt használhatnánk még be kell kötni a láncolt listába is, ami szintén egy atomi művelet segítségével történhet meg. A képponthoz tartozó előre lefoglalt mező mutatóját ki kell kicserélni az új értékre és a régit, amely a lista korábbi kezdő eleme volt, az új terület mutatójába menteni. Ezután következik a kijelölt memóriaterületre történő írás, amely teljesen azonos a korábbiakkal.

A következő lépés az árnyalás és átlagolás. Ehhez, hasonlóan az eddigiekhez egy téglalapot kell kirajzolni, ami lefedi a teljes képernyőt. A fragment shaderben azonnal végre lehet hajtani az árnyalást a képponthoz tartozó láncolt lista első elemén, ami előre le van foglalva és mindig létezik. A folytatást egy ciklus végzi el, ami bejárja a listát és minden elemét árnyalja. (A mélység értéket a mélység bufferből lehet megszerezni, amihez a mutatóban eltárolt minta indexre van szükség.)

Az eredményeket súlyozza a lefedett minták száma alapján és összegzi. Az elsőnek kiszámolt érték súlyát a végén kapjuk meg: ez a maximális mintaszám és a bejáráskor talált mezők össz mintaszámának különbsége lesz. Az eredményt osztani kell a maximális mintaszámmal, majd kiírni képernyőre.

Ez a módszer is megvalósítható egy renderpass és három subpass segítségével, mivel mindegyik lépésben a fragment shader csak az adott képponthoz tartozó értékeket használja fel.

6 A mérési folyamat

Az implementációkat 2 különböző Nvidia GPU-n vizsgáltam meg az alábbi tulajdonságokkal rendelkező platformokon:

GPU	OS	Driver verzió	Vulkan SDK	Fordító program
GTX 970	Windows 10 1903	436.48	1.1.121.2	G++ 9.2.0 (Mingww64)
GTX 1050	Windows 10 1903	440.97	1.1.121.2	VS 2019 16.3.6

6.1. táblázat: A mérésekhez használt rendszerek tulajdonságai.

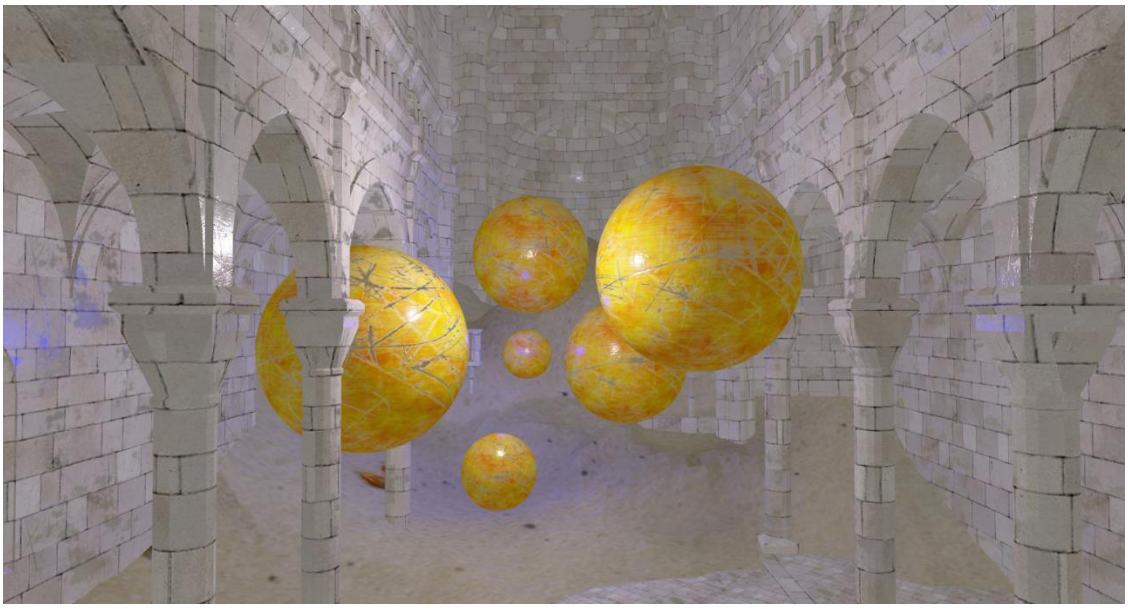
A mérések során 9 színteren vizsgáltam meg a korábban bemutatott 5 algoritmust. Az MSAA-t alkalmazó 3 technika esetében a 2x-es, 4x-es és 8x-os mintavételi frekvencia mellett is elvégeztem a méréseket. Ezek során a memória használatot, egy képkocka elkészítéséhez szükséges időt és az algoritmusokat alkotó, jól elhatárolható pass-ok lefutási idejét figyeltem. Ilyen pass volt a Z-prepass, geometry pass, lighting pass és a post-process pass. Ez utóbbira csak az FXAA végrehajtása során volt szükség.

6.1 Színterek

Az alábbiakban bemutatott három színtéren végeztem méréseket. Mindegyiken megvizsgáltam egy alacsony, egy közepes és egy magas számú pont fényforrást tartalmazó beállítást is. A színterek között különbség a komplexitás és az elhelyezett modellek vertex száma volt. A komplexitás alatt az egymást metsző, részben kitakaró háromszögek számát értem. Ezt azért fontos vizsgálni, mivel ezek miatt jelenik meg a geometriából adódó aliasing.

6.1.1 1. Színtér

Ennek a színtérnek a célja az volt, hogy egy relatív alacsony számú vertexet tartalmazó helyzetben vizsgáljam meg az algoritmusokat, ahol a komplexitás is alacsony. (A gömbök, illetve az épület felülete a legtöbb helyen sima, kevés a geometria által generált élek száma.)



6.2. ábra: Az 1. színtér felépítése és kamera pozíciója. Háromszögek száma: 1.083.115, vertexek száma: 1.424.145.

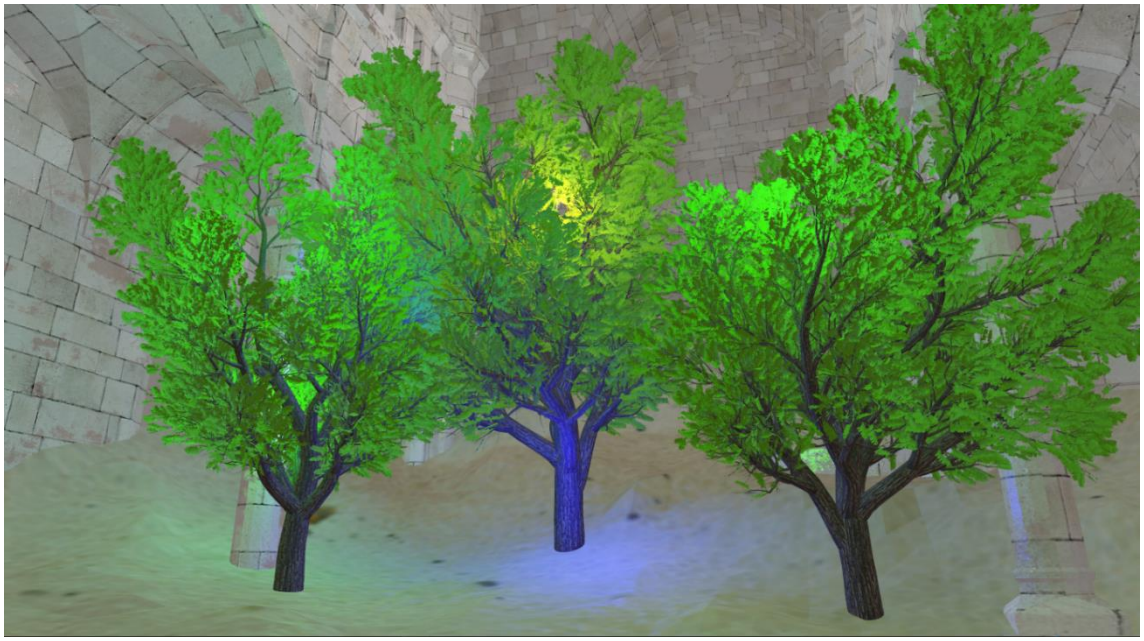
6.1.2 2. Színtér

Ennek a színtérnek a célja, az előzőhöz képest magasabb vertex szám vizsgálata volt, egy olyan beállítás mellet, ahol a komplexitás is nagyobb, de a képernyő csak egy kisebb részén összpontosul. (A fa leveleit alkotó háromszögek sokszorosan átfedik és metszik egymást.)

6.3. ábra: Az 1. színtér felépítése és kamera pozíciója. Háromszögek száma: 3.508.074, vertexek száma: 8.699.022.

6.1.3 3. Színtér

A színtér célja a sok vertex melletti teljesítmény vizsgálata volt úgy, hogy a képernyő nagy részén, nagy számban jelentek meg az elsimítandó élek.



6.4. ábra: Az 1. színtér felépítése és kamera pozíciója. Háromszögek száma: 8.548.456, vertexek száma: 23.820.168.

Az alacsony fényforrás szám 5, a közepes 20, a magas 50 pont fényforrást jelentett. Ezek a számok alacsonynak tűnhetnek népszerű deferred megjelenítők által támogatott több száz fényforráshoz. Ennek az az oka, hogy ezek a fényforrások mindig, erősségtől függetlenül az összes képpontot megvilágították. Azért nem alkalmaztam az ehhez kapcsolódó optimalizációkat, mert azzal csupán az árnyalások számát csökkentettem volna. Ez az egyes élsimítások esetén ugyanúgy befolyásolta volna a teljesítményt. Másképpen fogalmazva lehetett volna választani annyi optimalizált fényforrást, hogy ugyanazt a teljesítményt nyújtsák, mint az általam használt kevés nem optimalizált. Továbbá ezzel a megoldással könnyebben lehetett értelmezni a kapott eredményeket.

7 Eredmények összehasonlítása

A továbbiakban a kapott eredményeket fogom összehasonlítani. A nyers adatokat mennyiségük miatt itt nem mutatom be, ezek megtalálhatók lesznek a függelékben.

Az itt bemutatott adatok között a MSAA Z-prepass-val kombinált változata nem fog megjelenni, de az előbb említett függelékben megtalálható. Ennek az az oka, hogy memóriahasználat esetén nincs különbség a sima MSAA-hoz képest, futás idő tekintetében, pedig mindig alulmaradt a javasolt algoritmussal szemben, ami a várható eredmény volt. Valójában tehát teljesítette a feladatát, azaz annak az esetnek a kizárást, hogy a javasolt algoritmus csak a Z-prepass miatt teljesít jobban.

7.1 Memóriahasználat

Az alábbi táblázatban szereplő értékek 1920x1080-as felbontás mellett megabyte-ban értendők. A 2x, 4x, 8x az egy pixelhez tartozó minták számára utal. A javasolt algoritmus esetén nincsen fix memória méret csak egy minimum érték, ami szükséges a kép élsimítás nélküli helyes megjelenítéséhez és egy maximum érték, aminél több memóriára sosem lesz szükség. A táblázat a nem releváns (mindenütt azonos) memória foglalásokat nem tartalmazza. Ilyenek például a textúrák vagy a fényforrások adatai.

	No AA	FXAA	MSAA			A javasolt algoritmus					
			2x	4x	8x	2x		4x		8x	
						Min	Max	Min	Max	Min	Max
G-Buffer	23.73	23.73	47.46	94.92	189.84	27.69	55.37	27.69	110.74	27.69	221.48
Z-Buffer	7.91	7.91	15.82	31.64	63.28	15.82	15.82	31.64	31.64	63.28	63.28
Egyéb	0	7.91	0	0	0	0	0	0	0	0	0
Összesen	31.64	39.55	63.28	126.56	253.12	43.51	71.19	59.33	142.38	90.97	284.76

7.1. táblázat: A memóriahasználat megabyte-ban.

Összehasonlítva a javasolt algoritmust és az egyszerű MSAA-t látszik, hogy az utóbbi által igényelt memória mindig a minimum és maximum értékek közé esik. Az maximális eltérés 8x-os mintavételezés esetén 30MB körül van. Az, hogy a javasolt technika esetén mekkora memória méretet kell megválasztani az teljesen igény függő ezért ezek az értékek közvetlen nem nyújtanak túl sok információt. Emiatt érdemes megvizsgálni egy konkrét esetet.

7.2. ábra: A teljes élsimításhoz szükséges memória méréséhez használt kamerabeállítás.

Ebben a kamera beállításban a képernyő nagy részén a fa levelei és ágai látszódnak, amelyek sok elsimítandó élet generálnak. Azonban egy ilyen komplex helyzetben is (8x-os mintavételezés esetén) valójában a lefoglalt memóriának csak a töredéke lesz ténylegesen felhasználva. Emiatt ez a kép 52.03 MB méretű G-Buffer felhasználásával készült és ugyanazt az eredményt nyújtotta, mintha a maximális memóriát (221.48 MB) használtam volna. Ez összesen (Z-bufferrel együtt) 115.31 MB memóriát jelentett, ami alacsonyabb, mint a hagyományos 4x MSAA igénye. Tehát az új algoritmus potenciálisan jobb élsimítást tudott elvégezni kevesebb memória felhasználásával.

Érdemes még a 4x-es mintavételezést is megvizsgálni, hogy ott milyen eredményeket kapunk. Ebben az esetben 43.37 MB-os G-Bufferre, azaz összesen 75.01 MB memóriára volt szükség a teljes élsimításhoz. Ez az érték, ahogy várható volt a 4x MSAA memória igénye alatt van, de már nem sikerült a 2x-es alá menni.

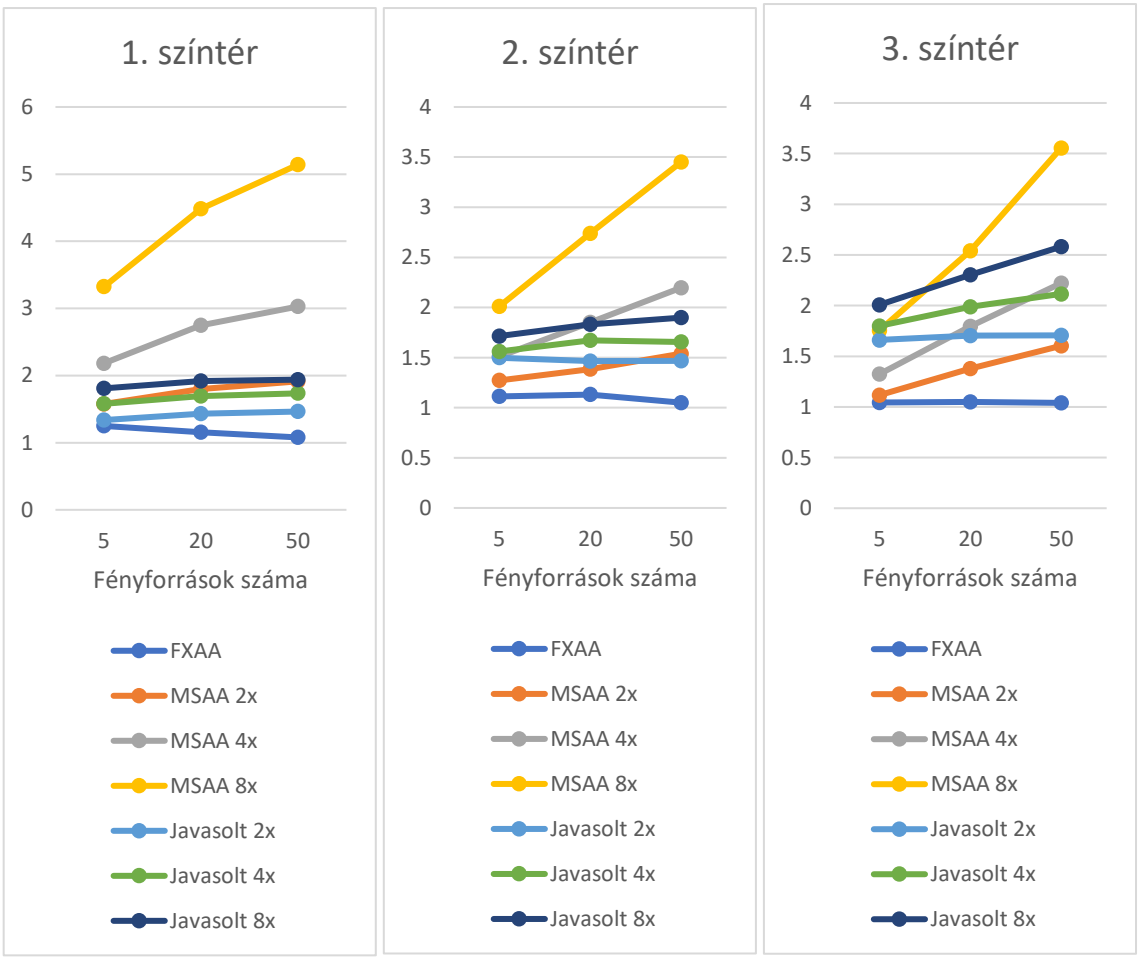
Gyakran a 8x-os élsimítás hatása alig-alig észrevehető a 4x-eshez képest, viszont a memória igénye, ahogy a táblázatban is látszik drasztikusan megnő. Emiatt ritkán szokták alkalmazni. Viszont ezzel az új módszerrel lehetőség nyílik arra, hogy megtartsuk a 8x-os élsimítást azokon a helyeken, ahol tényleg szükség van rá, míg a memória igény potenciálisan alacsonyabb lesz, mint a hagyományos 4x-es esetben.

Könnyen lehet azt is detektálni, ha az alkalmazás ki csúszik a memória korlátok közül. Ebben az esetben le lehet foglalni nagyobb memória területet és pár képkockát

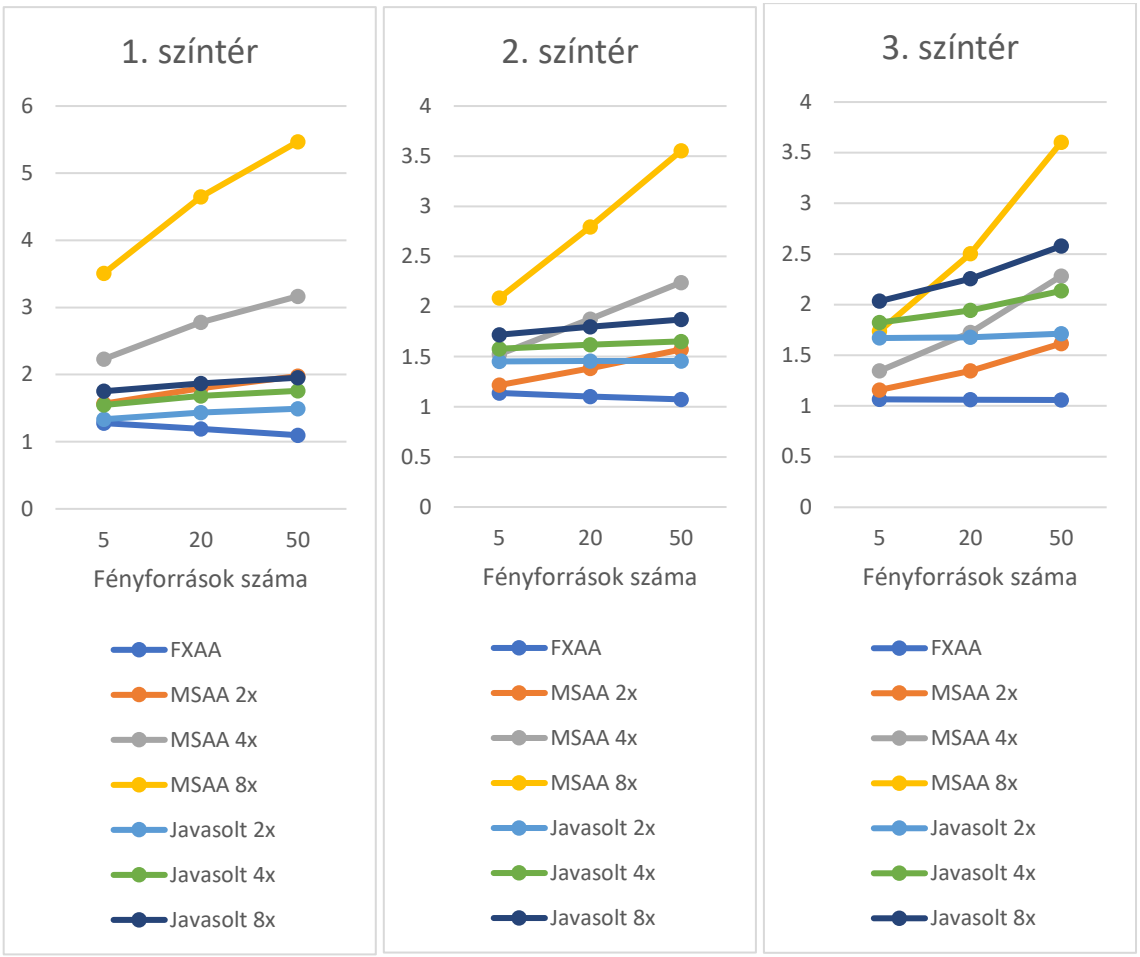
követően helyreállítható a teljes élsimítás. Hasonlóan, ha látszik, hogy konzisztensen a lefoglalt memória alatt maradunk, akkor a felesleges területet fel lehet szabadítani. Az, hogy több memória van lefoglalva, mint amennyire szükség lenne, a sebességet semmilyen módon nem befolyásolja.

7.2 Futás idő

Az összehasonlítás során először a futási időt, az élsimítás nélküli deferred shading futási idejéhez viszonyítottam. Ez azt jelenti, hogy az alábbi diagrammokon a függőleges tengely jelzi, hogy az adott algoritmus futási ideje hányszorosa volt az élsimítás nélküli változatnak. Lehet úgy is fogalmazni, hogy az élsimítás relatív költségét mutatják az ábrák. Ezzel a mértékkel az eltérő GPU teljesítményből és színtér komplexitásból adódó nagyságrendi különbségek megszűntek, és könnyebben lehet rávilágítani, hogy az egyes algoritmusok hogyan viselkednek egy adott helyzetben.



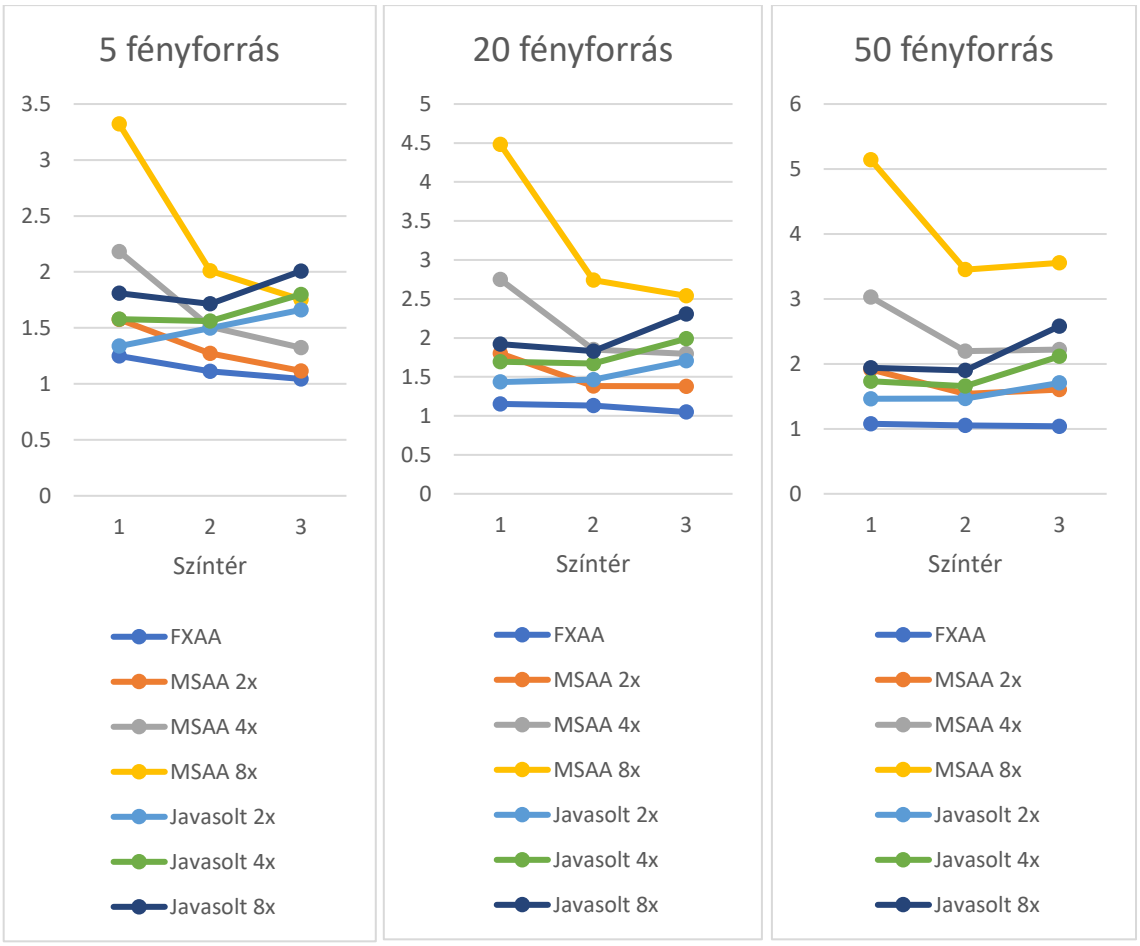
7.3. ábra: Az élsimítás nélküli változathoz képesti relatív futásidő GTX 970-en. Ez a nézet fix színtér mellett mutatja a fényforrás szám változásának hatását.



7.4. ábra: Az élsimítás nélküli változathoz képesti relatív futásidő GTX 1050-en. Ez a nézet fix színtér mellett mutatja a fényforrás szám változásának hatását.

Azonnal megfigyelhető, hogy az FXAA teljesítménye javul a fényforrások számának növelésével. Ennek az az oka, hogy a futási ideje nem függ a színtér komplexitásától. Emiatt, ahogy nő a futási idő úgy lesz egyre kisebb a relatív költsége. Ez a későbbi diagrammok során is hasonló lesz ezért nem fogom többször kiemelni.

Mindkét GPU esetében látszik, hogy az MSAA rosszabbul reagál (meredekebb a változás) a növekvő fényforrás (azaz árnyalás) számra, mint a bemutatott algoritmus. Ahogy növekszik a fényforrások száma úgy egyre több és több felesleges árnyalást kell végrehajtania. Ezek színterenként eltérő ütemben, de csökkentik a sebességét. Emiatt a javasolt algoritmus 20 fényforrástól kezdve már a 3. színtéren is jobban tud teljesíteni.



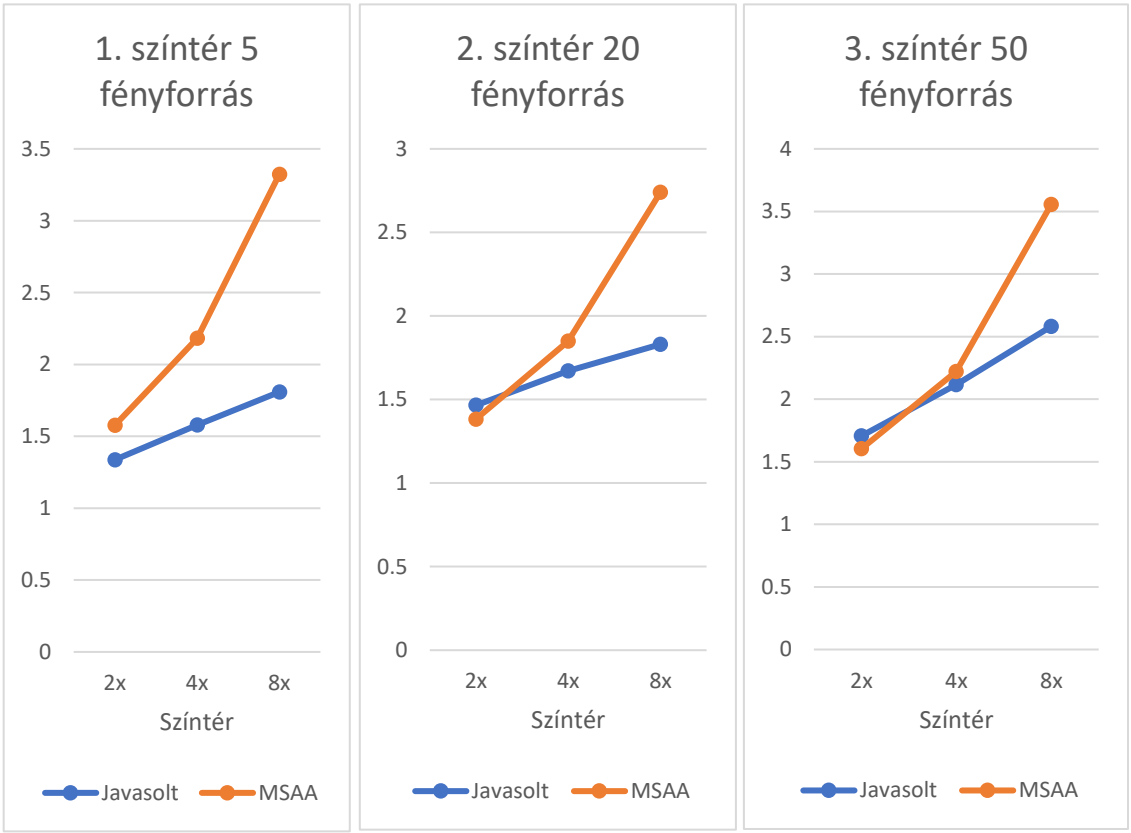
7.5. ábra: Az élsimítás nélküli változathoz képesti relatív futásidő GTX 970-en. Ez a nézet fix fényforrás szám mellett mutatja a színtér komplexitás változásának hatását.

Ezek a diagrammok ugyanazokat az adatok ábrázolják, mint az előző 3 csak más nézőpontból. Itt a színtér fényforrásainak számát választottam konstansnak és a geometriai komplexitás függvényében látható a relatív teljesítmény.

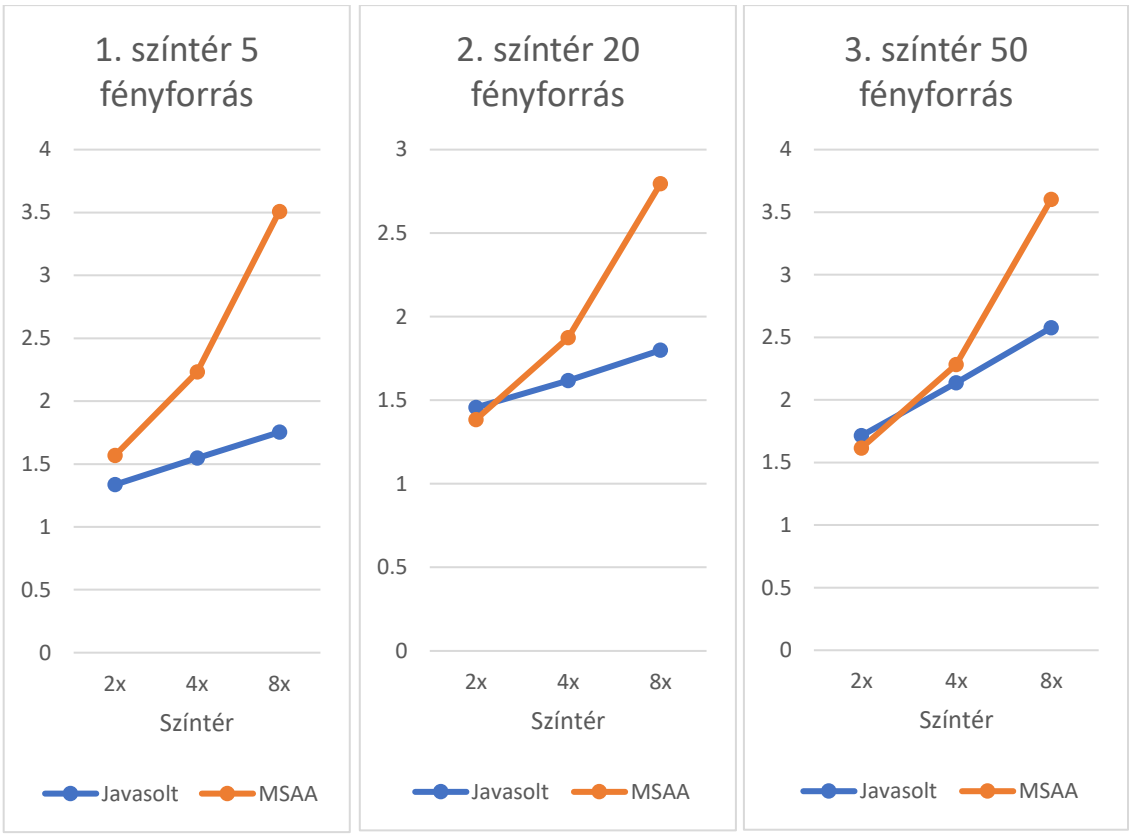
A hagyományos MSAA esetében egy csökkenő tendencia figyelhető meg az 1. és 2. szintér között. Az MSAA fő problémája az árnyalások száma. Ha a fényforrások számát (ami ezt közvetlen befolyásolja) konstansnak választjuk, akkor a kirajzolt geometria mennyiségének növelése a relatív teljesítmény pozitívan befolyásolja. (Az elsimítandó élek száma nem növekedett drasztikusan az 1. és 2. színtér között.) A 2. és 3. színtér között a csökkenés kisebb mértékű, sőt 50 fényforrás esetén egyenesen nő a relatív költség. Ez azért van így, mivel itt nem csak a GPU-ra felküldött geometria mennyisége nőtt, hanem az elsimítandó élek száma is. Ez megnöveli az árnyalások számát, ami 50 fényforrás esetén lesz a legjobban észrevehető.

A javasolt megoldás esetében más a helyzet. Itt az 1. és 2. színtér között közel állandó marad az ábrázolt érték, a 2. és 3. között pedig növekszik. Az 1. és második 2.

színtér esetén a felküldött geometria növekedését kompenzálja, hogy a Z-prepass speciálisan (gyorsabb feldolgozással) van implementálva a modern GPU-kban. A szükséges árnyalások száma sem változik nagy mértékben az 1. és 2. eset között. A 2. és 3. között viszont már nem rejti el a geometria két alkalommal történő kirajzolását a GPU az egyre nagyobb mennyiség miatt. Az árnyalások száma is növekszik a kamera beállítás következtében.



7.6. ábra: Az élsimítás nélküli változathoz képesti relatív futásidő GTX 970-en. Ez a nézet fix színtér és fényforrás szám mellett mutatja az 2x-es, 4x-es és 8x-os mintavételi frekvencia teljesítményét.



7.7. ábra: Az élsimítás nélküli változathoz képesti relatív futásidő GTX 1050-en. Ez a nézet fix színtér és fényforrás szám mellett mutatja az 2x-es, 4x-es és 8x-os mintavételi frekvencia teljesítményét.

Ezeken a diagrammokon jól látszik, hogyan reagálnak az algoritmusok a minták számának növelésére. A hagyományos MSAA relatív költsége jóval meredekebb ütemben növekszik, mint a javasolt módszernek. Az MSAA rosszul reagál a felesleges mintákra, sok felesleges árnyalás kerül végrehajtásra csak azért, hogy a képernyő egy kis részén még jobb legyen az élsimítás. Ezzel szemben a javasolt algoritmus soha nem hajt végre felesleges árnyalást, így a minták számának növelése csak ott fejti ki hatását, ahol tényleg szükség van rá. Az is megfigyelhető, hogy a 2. és 3. diagrammon 2x-es élsimítás mellett magasabb a kezdeti relatív költsége, mint a hagyományos MSAA-nak. Ezt az algoritmus plusz adminisztrációs költsége okozza, amire szükség van az adatstruktúra létrehozásához (Z-prepass) és karbantartásához. Azonban ez a plusz költség teszi lehetővé, hogy több minta vétele ne legyen rá akkora hatással.

A két GPU esetében nemcsak a tendencia, de a konkrét értékek is közel azonosak, ami még jobban alátámasztja a fenti következtetéseket.

Érdemes a relatív adatok mellett a konkrét értékeket is megvizsgálni néhány helyzetben.

	MSAA 4x	Javasolt élsimítás 4x	Javasolt élsimítás 8x
1. színtér – 5 fényforrás	5.2648	3.8092	4.3629
1. színtér – 50 fényforrás	21.987	12.5951	14.0772
2. színtér – 5 fényforrás	7.9021	8.1721	8.97702
3. színtér – 5 fényforrás	14.069	19.1354	21.3446
3. színtér – 50 fényforrás	33.293	31.7038	38.6841

	MSAA 4x	Javasolt élsimítás 4x	Javasolt élsimítás 8x
1. színtér – 5 fényforrás	9.87546	6.84544	7.7527
1. színtér – 50 fényforrás	43.3121	24.0456	26.7233
2. színtér – 5 fényforrás	14.7917	15.3364	16.6902
3. színtér – 5 fényforrás	26.6056	36.0591	40.2319
3. színtér – 50 fényforrás	65.9231	61.6878	74.4489

7.8. táblázat: Konkrét mérési eredmények a futásidőről fent GTX 970-en, lent GTX 1050-en, ezredmásodpercben (ms).

Az 1. színtér esetén mind alacsony, mind magas számú fényforrás esetén a javasolt technika nem csak gyorsabb, mint az MSAA, de a 8x-os mintavételezést is hatékonyabban hajtja végre, mint a hagyományos változat a 4x-eset. Ez a 2. és 3. színtér esetén a nagyobb mennyiségű geometria miatt nem teljesül, de a táblázat utolsó sorából látszik, hogy elég sok fényforrás esetén le tudja hagyni teljesítményben a sima MSAA-t azonos frekvenciájú mintavételezés esetén.

A GTX 1050 esetén a táblázat második sorában látható, hogy míg a hagyományos MSAA nem volt elég gyors, például egy valós idejű játék élvezhető futtatásához, az új megoldás képes volt jobb élsimítás mellett elfogadható színtű teljesítményre.

Egy képkocka elkészítését is meg lehet vizsgálni részletesebben, azaz, hogy melyik fázis mennyi időt vett igénybe.

1. színtér – 5 fényforrás	Z-prepass	Geometry pass	Lighting pass
MSAA 4x	0	2.9944	2.2704
Javasolt élsimítás 4x	0.7012	1.8591	1.2487

1. színtér – 50 fényforrás	Z-prepass	Geometry pass	Lighting pass
MSAA 4x	0	2.9949	18.992
Javasolt élsimítás 4x	0.91801	2.0902	11.069

3. színtér – 5 fényforrás	Z-prepass	Geometry pass	Lighting pass
MSAA 4x	0	11.429	2.6397
Javasolt élsimítás 4x	6.89718	10.4294	1.8088

3. színtér – 50 fényforrás	Z-prepass	Geometry pass	Lighting pass
MSAA 4x	0	11.061	22.232
Javasolt élsimítás 4x	6.54416	10.3358	14.8238

7.9. táblázat: Lépésekre bontott futásidő GTX 970-en, ezredmásodpercben (ms).

Az 1. színtér esetében a Z-prepass gyakorlatilag nem ront a teljesítményen. A korábbi eredményekben is ezért teljesített ilyen jól a javasolt algoritmus ebben a jelenetben. A 3. színtér esetében már más a helyzet, 5 fényforrásnál a Z-prepass miatti hátrányt még nem tudja „ledolgozni” a lighting pass során. Azonban 50 fényforrás esetén megfordul a helyzet és körülbelül 2 ms-al gyorsabb lesz a javasolt algoritmus.

8 Összefoglalás

Dolgozatomban egy újszerű megközelítést mutattam be a deferred shading és MSAA együttes alkalmazásának problémájára. Az eddig ismert megoldásokkal több szempontból, összesen 9 színtér segítségével hasonlítottam össze több különböző hardveren. A mérésekhez egy C++ és Vulkan alapú keretalkalmazást készítettem, mely lehetővé tette 3 különböző élsimító (és 2 referencia) algoritmus hatékony implementálását, illetve felgyorsította a tesztelés és teljesítmény elemzés folyamatát.

A kapott eredmények alapján arra a következtetésre jutottam, hogy az új módszer képes alacsonyabb memória használat mellett, gyorsabban, jobb élsimítást elérni, mint az eddig ismert MSAA alapú megoldások.

A memóriahasználata az új technikának rugalmasan paraméterezhető a színtér, a rendelkezésre álló memória és egyéni igény szerint teljesítmény csökkenés nélkül. Jobban reagál a mintavételezési frekvencia és a fényforrás szám növelésére, minta az eddigi módszerek. Hátrányt a sok vertexből álló színterek jelentenek, de az eredmények alapján ez csak szélsőséges esetben jelent problémát.

Ezek mellett egy rugalmasabb rendszert is képez, ami lehetővé teszi modern technológiák (VRS) részleges használatát régebbi hardveren is. A rendszer továbbá potenciálisan optimálisabb lehetőséget biztosít egyéb technikák ötvözésére.

9 További lehetőségek

Az alábbiakban, néhány idő hiányában egyelőre nem implementált ötletet mutatok be, amelyek tovább javíthatnának a javasolt módszer teljesítményén, avagy új funkcionalitást adnának hozzá.

9.1 Memória foglalás feldarabolása

Mivel minden képpont ugyanazt az egy számlálót növeli atomikusan, ezért valamikor a hardvernek sorosítania kell ezeket a kéréseket. Ezen részben már sikerült javítani azzal, hogy az amúgy is szükséges területet nem foglaljuk le külön. Viszont komplex színterek esetén, ahol sok helyen lesz szükség kettő vagy több mintára, a probléma továbbra is fennáll.

A modern GPU-k gyakran tile-based rendering architektúrát alkalmaznak (főleg mobil eszközökön), ami azt jelenti, hogy egyszerre nem a teljes képernyőre rajzolnak, hanem csak egy kisebb darabjára. Ennek az egyik oka, hogy így kevesebb belső gyors memóriára van egyszerre szükség. Ez a technika asztali GPU-kban is megtalálható (pl.: Nvidia Maxwell architektúrától kezdve), amit ebben az esetben ki lehet használni.

Az az ötlet, hogy a teljes memóriaterületet fel lehet darabolni úgy, hogy a párhuzamosan számolódó minták ne zavarják egymást a foglaláskor. Például a képet képzeletben 32x32-es blokkokra osztjuk, amelyek a tile based architektúrát próbálják követni. Ekkor egy ilyen blokkban bármelyik képpont feldolgozása futhat a másikkal párhuzamosan. Ezért minden képpontnak egy egyedi memóriaterületre (és számlálóra) van szüksége, amiből foglalhat. De két 32x32-es blokk feldolgozása soha nem fog párhuzamosan zajlani, ezért a többi blokk használhatja ezeket a memóriaterületeket. Így elméletben soha nem lesz ütközés a számláló növelésekor. Gyakorlatban mégis előfordulhat ütközés, mivel nem garantált az a viselkedés, amire ez a módszer épít, ezekben az esetekben az atomi műveletek továbbra is megoldják a problémát.

Egy azonnal felvetődő probléma, hogy így nehezebb lesz optimalizálni a blokkok maximális számát, mivel könnyen lehet, hogy az egyik területen marad használatlan blokk, míg máshol elfogy az összes, de továbbiakra is szükség lenne. Egy kicsit javít a helyzeten, hogy azok a területek, ahova “nem jutott” élsimítás nem blokkosan

helyezkednek el (mint e nélkül a módszer nélkül), hanem elszórtan a teljes képen, kevésbé feltűnően.

Plusz egy indirekcióval elméletben ezen lehetne tovább javítani. Minden memória foglaló egység kapna egy kis blokkot az induláskor, amit felhasználhat, de nincs kiosztva az összes memóriaterület. Ha egy blokk megtelik, akkor a tartalék területről lehet lefoglalni egy új blokkot. Ez megoldja a memória méret problémáját, de az implementációhoz nem lesznek elegek az egyszerű atomi műveletek, hanem zárakat is meg kell valósítani a segítségükkel, hogy telítődés esetén csak egy új blokk legyen lefoglalva. Ez valószínűleg már nem lesz hatékony. Új paraméterként a foglalási egység mérete is megjelenik, aminek az optimalizálása a blokk méret mellett egy további kihívás lesz.

9.2 Változó mintaszám

A maximális lehetséges mintavételezési frekvenciát a mélység buffer nagysága (multisample beállítása) határozza meg. Azonban ez alá szabadon lehet menni egészen a képpontonkénti 1db mintáig. Ezt akár objektumonként is meg lehet tenni (például ha tudjuk, hogy egy objektum olyan helyen lesz csak, ahol nem igényel élsimítást), illetve a VRS-hez hasonlóan a képernyő különböző részein is lehet eltérő nagyságú multisampling-ot alkalmazni. Ehhez nem textúrában kell megadni a paramétereket, hanem magában a geometry pass fragment shaderének kódjában kell kiválasztani a különböző tartományokat.

9.3 Z-Prepass megszüntetése

A legnagyobb teljesítmény problémát a Z-prepass jelenti a javasolt algoritmusban. Erre azért van szükség, hogy a teljesen kitakart, de időben korábban kirajzolt háromszögek ne használjanak feleslegesen memóriát. A cél az lenne, hogy ezt a helyzetet Z-prepass nélkül a shaderen belül tudjuk hatékonyan kezelni.

Arra van szükség, hogy a fragment shader-ben tudjuk detektálni, hogy az aktuális fragmensel mikor írunk felül egy korábbit (vagy korábbiakat), ezeket el tudjuk távolítani az adatszerkezetből és ha ezután találunk egy nem használt blokkot, akkor azt használjuk fel memória foglalás helyett.

Ha a láncolt listás megközelítésnél maradunk, akkor el kell tárolni minden blokkban, hogy az pontosan melyik fragmenseket reprezentálja. Ezután a listát minden

alkalommal végig kell járni és módosítani az eltárolt lefedettséget. A kiürült blokkokat eközben lehet azonosítani és megspórolni a memória foglalást az egyik felhasználásával. Ha mégis szükség lenne rá, akkor az új blokkot a lista végére lehet bekötni. Ez azt is jelenti, hogy nem szüntettünk meg minden “memória szivárgást”, mert ha egy blokkot egy pixelhez lefoglaltunk, akkor utána azt csak ott tudjuk felhasználni. Azt a kérdést kell megvizsgálni, hogy mi történik, ha két shader futás egyszerre ugyanazt a listát manipulálja. Mivel a lista struktúrája nem változik, ezért a bejárást nem fogják akadályozni. A lefedettség módosítása és az üres blokkok megszerzése megoldható atomi műveletekkel. A problémát az fogja jelenteni, hogy ha a lista bejárása közben a később beérkezett fragmens “megelőzi” a másikat. Erre lehet megoldás a *fragment_shader_interlock* kiegészítés használata. Ezzel két shader lefutás kölcsönösen kizáróvá és rendezetté tehető a kód egy darabjában.

Látható, hogy egy ilyen algoritmus implementálásához sok dinamikus feltételre van szükség, illetve a hardvernek is sorosítani kell bizonyos shader lefutásokat. Emellett még a teljes láncolt listát is be kell járni, ami potenciálisan 8 különböző memóriaterület olvasását jelenti. Emiatt meggondolandó a láncolt lista megszüntetése és a mutatók, illetve lefedettségek egy helyen tárolása. Ezzel ugyan nő a memóriahasználat, de a láncolt lista bejárásához elég lesz egy kicsi folytonos memóriaterületről olvasni.

9.4 Platform bővítés

Az új módszert dolgozatom keretein belül csak a rendelkezésemre álló Windows 10 operációs rendszeren Nvidia GPU-k segítségével teszteltem. A még teljesebb eredményekhez érdemes lehet egyéb rendszereket (platformokat) és más gyártóktól származó hardvereket is bevonni a további vizsgálatokba.

Irodalomjegyzék

[1] **Lottes, Timothy** (2009). *FXAA. NVIDIA.*

[2] **Reshetov, Alexander** (2009). *Morphological Antialiasing.* 2009 ACM 978-1-60558-603-8/09/0008.

[3] **Jimenez, Jorge** (2012). *SMAA: Enhanced Subpixel Morphological Antialiasing.* Eurographics 2012 Vol. 31, No. 2.

[4] **NVIDIA**. *Antialiased Deferred Rendering*.

[5] **NVIDIA** (2018). *Turing GPU Architecture*.

[6] **R. Cook and K. Torrance.** *A reflectance model for computer graphics.* Computer Graphics (SIGGRAPH '81 Proceedings), Vol. 15, No. 3, July 1981, pp. 301–316.

[7] **Reinhard, Erik** (2002). *Photographic tone reproduction for digital images.* ACM Transactions on Graphics. 21 (3).

[8] **Weisstein, Eric W.** *Lambert Azimuthal Equal-Area Projection.* MathWorld - A Wolfram Web Resource. http://mathworld.wolfram.com/LambertAzimuthalEqual-AreaProjection.html (utoljára megtekintve: 2019. 10. 27.)

Részletes mérési eredmények – GTX 970

Az alábbi táblázat a nyers eredményeket tartalmazza. Minden érték ezredmásodpercben (ms) értendő. Az üres cellák azt jelentik, hogy a lépés nem létezik az algoritmusban.

		Deferred	Deferred FXAA	Deferred MSAA			Deferred MSAA (Z-prepass)			Deferred + Javasolt		
Színtér.Fényforrás	Lépés			2X	4X	8X	2X	4X	8X	2X	4X	8X
1.5	Z-prepass						0.6403	0.7437	1.0185	0.5412	0.7012	0.91308
	Geometry pass	1.8091	1.7626	2.3987	2.9944	3.9992	1.345	1.7909	2.2951	1.6264	1.8591	2.0809
	Lighting pass	0.6031	0.6252	1.4038	2.2704	4.0148	1.3756	2.2825	4.0231	1.0572	1.2487	1.3688
	Postprocess pass		0.6258									
	Frame time	2.4123	3.0137	3.8026	5.2648	8.0141	3.3609	4.8172	7.3367	3.2249	3.8092	4.3629
1.20	Z-prepass						0.60502	0.7444	1.0114	0.5442	0.7024	0.9146
	Geometry pass	1.7925	1.7563	2.4035	3.0828	3.9922	1.4239	1.894	2.3002	1.6225	1.845	2.0856
	Lighting pass	2.1723	2.1918	4.7362	7.8181	13.784	4.74672	7.8214	13.7665	3.5177	4.1738	4.6119
	Postprocess pass		0.6253									
	Frame time	3.9649	4.5735	7.1398	10.901	17.776	6.7756	10.46	17.0782	5.6844	6.7213	7.6122
1.50	Z-prepass						0.58982	0.75395	1.01062	0.5343	0.68486	0.91801
	Geometry pass	1.91949	1.8571	2.3942	2.9949	3.9779	1.4407	1.7942	2.29053	1.6235	1.8443	2.0902
	Lighting pass	5.3415	5.3437	11.501	18.992	33.349	11.5186	18.9285	33.3317	8.4682	10.0659	11.069
	Postprocess pass		0.62627									
	Frame time	7.26099	7.8272	13.895	21.987	37.327	13.5491	21.4767	36.6328	10.6262	12.5951	14.0772
2.5	Z-prepass						2.3979	2.6037	2.9427	2.3304	2.5352	2.8397
	Geometry pass	4.63171	4.5705	5.1339	6.0066	7.3046	4.09491	4.4734	5.04563	4.5435	4.4772	4.7857
	Lighting pass	0.60435	0.62534	1.5263	1.8954	3.2227	1.5257	1.8868	3.23395	0.96902	1.1597	1.3515
	Postprocess pass		0.6288									
	Frame time	5.236	5.8246	6.6602	7.9021	10.527	8.0186	8.9639	11.2224	7.84301	8.1721	8.97702
2.20	Z-prepass						2.3784	2.58531	2.9247	2.3329	2.5343	2.92314
	Geometry pass	4.54163	4.4753	5.0657	5.8627	7.1601	4.1325	4.4651	5.03005	4.25277	4.79376	4.80531
	Lighting pass	2.17523	2.1979	4.221	6.5632	11.241	4.21482	6.57053	11.2194	3.25648	3.9002	4.5672
	Postprocess pass		0.92115									
	Frame time	6.71686	7.5944	9.2868	12.426	18.401	10.7259	13.621	19.1742	9.84214	11.2284	12.2956
2.50	Z-prepass						2.39098	2.5828	2.93773	2.4215	2.62301	2.8499
	Geometry pass	4.61539	4.49078	5.0581	5.8568	7.2279	4.10576	4.4499	5.03117	4.3649	4.48598	5.0881
	Lighting pass	5.3489	5.3455	10.274	16.031	27.157	10.1924	16.0048	27.1356	7.83888	9.40138	10.979
	Postprocess pass		0.63059									
	Frame time	9.9642	10.4669	15.332	21.888	34.385	16.6892	23.0376	35.1045	14.6253	16.5104	18.9172
3.5	Z-prepass						6.49126	6.7516	7.51197	6.3759	6.89718	7.58394
	Geometry pass	9.7172	9.87098	10.269	11.429	13.634	9.7526	10.4526	12.4138	10.1153	10.4294	11.3473
	Lighting pass	0.915104	0.61971	1.5981	2.6397	5.0103	1.6031	2.64406	4.68541	1.17165	1.8088	2.41334
	Postprocess pass		0.61136									

	Frame time	10.6324	11.102	11.867	14.069	18.644	17.847	19.8483	24.6111	17.6629	19.1354	21.3446
3.20	Z-prepass						6.35094	6.95488	7.67699	6.12614	6.84064	7.6434
	Geometry pass	9.6415	9.5854	10.7	12.012	13.655	9.909	10.4871	12.3463	10.047	10.4558	11.3336
	Lighting pass	2.15606	2.17789	5.5809	9.176	16.307	5.59715	9.16422	16.3627	3.9395	6.16189	8.21482
	Postprocess pass		0.61513									
	Frame time	11.7976	12.378	16.281	21.188	29.961	21.8572	26.6062	36.386	20.1127	23.4584	27.1919
3.50	Z-prepass						6.20608	6.8543	7.49101	6.20714	6.54416	7.36746
	Geometry pass	9.66435	9.6236	10.461	11.061	13.356	9.75229	10.5307	12.2286	9.85578	10.3358	11.4365
	Lighting pass	5.31958	5.33782	13.565	22.232	39.927	13.7271	22.2978	39.4426	9.50355	14.8238	19.8801
	Postprocess pass		0.616576									
	Frame time	14.9839	15.578	24.026	33.293	53.283	29.6855	39.6828	59.1622	25.5665	31.7038	38.6841

Részletes mérési eredmények – GTX 1050

Az alábbi táblázat a nyers eredményeket tartalmazza. Minden érték ezredmásodpercben (ms) értendő. Az üres cellák azt jelentik, hogy a lépés nem létezik az algoritmusban.

		Deferred	Deferred FXAA	Deferred MSAA			Deferred MSAA (Z-prepass)			Deferred + Javasolt		
Színtér.Fényforrás	Lépés			2X	4X	8X	2X	4X	8X	2X	4X	8X
1.5	Z-prepass						0.88064	1.11309	1.56877	0.85606	1.06086	1.43258
	Geometry pass	3.21843	3.22355	4.18509	5.2777	7.45165	2.4361	2.89075	3.98336	2.97779	3.33619	3.68128
	Lighting pass	1.20525	1.20525	2.74739	4.59776	8.0681	2.78016	4.5527	8.01485	2.0736	2.44838	2.63885
	Postprocess pass		1.22163									
	Frame time	4.42368	5.65043	6.93248	9.87546	15.5197	6.0969	8.55654	13.567	5.90746	6.84544	7.7527
1.20	Z-prepass						0.88269	1.11514	1.55853	0.85709	1.05882	1.43974
	Geometry pass	3.22765	3.43757	4.01818	5.08314	7.23149	2.39718	2.89382	3.97824	2.97677	3.36077	3.64442
	Lighting pass	4.26189	4.26394	9.42899	15.7153	27.5681	9.39827	15.7143	27.564	6.91098	8.19712	8.90061
	Postprocess pass		1.22778									
	Frame time	7.48954	8.92928	13.4472	20.7985	34.7996	12.6781	19.7233	33.1008	10.7448	12.6167	13.9848
1.50	Z-prepass						0.88474	1.11718	1.56365	0.86016	1.05882	1.43872
	Geometry pass	3.23277	3.33414	4.1257	5.10259	7.46086	2.44634	2.89178	3.94035	2.98086	3.32493	3.69869
	Lighting pass	10.4417	10.4806	22.9233	38.2095	67.3311	22.9775	38.1932	67.2963	16.556	19.6618	21.5859
	Postprocess pass		1.19091									
	Frame time	13.6745	15.0057	27.049	43.3121	74.7919	26.3086	42.2021	72.8003	20.3971	24.0456	26.7233
2.5	Z-prepass						4.26701	4.64486	5.20499	4.1472	4.52813	5.07187
	Geometry pass	8.50534	8.57702	9.37062	10.9599	13.7492	7.65952	8.29133	9.5785	8.07424	8.52173	8.9559
	Lighting pass	1.20627	1.24416	2.43814	3.83181	6.49011	2.4361	3.79187	6.45632	1.88314	2.28659	2.6624
	Postprocess pass		1.23085									
	Frame time	9.71162	11.052	11.8088	14.7917	20.2394	14.3626	16.7281	21.2398	14.1046	15.3364	16.6902
2.20	Z-prepass						4.25472	4.50765	5.17018	4.23629	4.60288	5.1712
	Geometry pass	8.5248	8.60672	9.36653	10.9793	13.697	7.6288	8.30464	9.55699	8.08653	8.47872	8.9815
	Lighting pass	4.25882	4.25984	8.33126	12.9833	22.0396	8.35584	12.9946	22.016	6.30272	7.60218	8.84941
	Postprocess pass		1.22982									
	Frame time	12.7836	14.0964	17.6978	23.9626	35.7366	20.2394	25.8068	36.7432	18.6255	20.6838	23.0021
2.50	Z-prepass						4.16768	4.5097	5.15994	4.23424	4.45542	5.07597
	Geometry pass	8.4951	8.63027	9.47814	10.9107	13.697	7.66464	8.29952	9.61229	8.07424	8.49306	8.96
	Lighting pass	10.367	10.4038	20.1646	31.3395	53.3453	20.1114	31.2771	53.1098	15.1685	18.217	21.2439
	Postprocess pass		1.2247									
	Frame time	18.8621	20.2588	29.6428	42.2502	67.0423	31.9437	44.0863	67.882	27.477	31.1654	35.2799
3.5	Z-prepass						11.3306	12.1518	13.5322	11.2333	12.0556	13.3857
	Geometry pass	18.5795	18.7095	19.6454	21.2593	24.9672	18.8764	20.0305	22.7666	19.5062	20.4841	22.1276
	Lighting pass	1.19808	1.19706	3.25325	5.3463	9.41773	3.23994	5.33402	9.47302	2.29683	3.51949	4.71859
	Postprocess pass		1.16531									
	Frame time	19.7775	21.0719	22.8987	26.6056	34.3849	33.4469	37.5163	45.7718	33.0363	36.0591	40.2319
3.20	Z-prepass						11.3295	12.034	13.4779	11.2323	11.9562	13.4851
	Geometry pass	18.7023	18.8815	19.7888	21.2941	24.9446	18.9921	20.14	22.8321	19.5645	20.5875	22.2362

	Lighting pass	4.25267	4.27725	11.094	18.3153	32.4659	11.1657	18.4453	32.7137	7.71174	12.0402	16.0584
	Postprocess pass		1.19398									
	Frame time	22.955	24.3528	30.8828	39.6093	57.4106	41.4874	50.6194	69.0237	38.5085	44.5839	51.7796
3.50	Z-prepass						11.2548	12.0781	13.5864	11.3736	11.9613	13.3837
	Geometry pass	18.5938	18.9706	19.7755	21.3883	25.0716	18.946	20.1175	22.7983	19.5666	20.5967	22.2341
	Lighting pass	10.2892	10.4202	26.8974	44.5348	78.9832	26.921	44.4344	78.9299	18.5569	29.1297	38.8311
	Postprocess pass		1.19808									
	Frame time	28.8829	30.5889	46.6729	65.9231	104.055	57.1218	76.63	115.315	49.4971	61.6878	74.4489