BUDAPESTI MŰSZAKI ÉS GAZDASÁGTUDOMÁNYI EGYETEM
Villamosmérnöki és Informatikai Kar
Irányítástechnika és Informatika Tanszék

Lágy árnyékok képtérbeli számítása valós időben

TUDOMÁNYOS DIÁKKÖRI KONFERENCIA
DOLGOZAT

Készítette: Tamás Márton
mérnök informatikus
szakos hallgató

Konzulens: Dr. Magdics Milán
adjunktus

Tudományos Diákköri Konferencia
Budapest, 2015

Tartalomjegyzék

1. ÁTTEKINTÉS

Az árnyékok fontos szerepet játszanak az objektumok közötti kapcsolatok meghatározásában, térbeli koherencia kialakításában, a jelenet kompozíciójának javításában, a kép kontrasztosságában és a képen kívüli területek jelzésében.

Gyakran használják játékmechanika részeként, arra, hogy falakra árnyakat vetítsenek ezzel új formákat létrehozva melyek egy teljesen más történetet mesélhetnek el. Az árnyékokat gyakran használják arra is, hogy vezessék a szemlélő figyelmét vagy eltakarjanak vele lényegtelen részleteket.

A számítógépes grafikában a fényeket gyakran reprezentálják egy pontszerű fényforrásként, térfogat nélkül. Ezek a matematikai fényforrások csak kemény szélű árnyékokat tudnak képezni (egy pont vagy árnyékban van, vagy nem), amit umbrának hívnak. A való életben azonban a fényforrásoknak van térfogata (mint a Napnak), és emiatt puha szélű árnyékokat képeznek, amiknek három részük van: az umbra, a penumbra (egy pont részlegesen árnyékban van) és az antumbra (az árnyék ilyen részéről az árnyék vetítő objektumot teljesen körbefogja a fényforrás, mint egy napfogyatkozásnál).

Az 1-es ábrán látható egy valós példa umbrára, penumbrára és antumbrára.

Ábra 1 Példa umbrára, penumbrára és antumbrára a való világból. Az objektumokat egy asztali lámpa világítja meg.

A dolgozat mögötti motiváció az volt, hogy a jelenlegi valós idejű 3D grafikában használatos algoritmusok nem képesek sok (több mint 1-2) fényforráshoz változó lágyságú árnyékokat számolni hatékonyan. Így ezeket az eljárásokat költségük miatt csak 1-2 fényforrásra szokták alkalmazni. Az itt bemutatott technika ehhez képest viszont képes akár több száz fényforrás változó lágyságú árnyékainak valós idejű megjelenítésére.

Ez a dolgozat egy új technikát ír le, amely kiegészíti az eredeti Screen Space Soft Shadows algoritmust [Gumbau10] annak érdekében, hogy a képszintézis sebessége optimális legyen és figyelembe vegyük az egymásra vetülő árnyékokat, illetve átlátszó árnyékokat. Az átlapolódó árnyékok képtérbeli szűrésére rétegeket vezetünk be.

A képtérbeli szűrésen alapuló technikák egyetlen hátránya, hogy a képtérbeli információ korlátoltsága miatt a szűrés képi hibákhoz vezethet. Ez szerencsére csak a szűrés eredményét befolyásolja, ettől függetlenül mindenhol lesz a végső képen árnyék, ahol kell lennie.

Az új technika célja az, hogy valós időben több száz valósághű, puha szélű árnyékot lehessen megjeleníteni. Továbbá az is célja, hogy könnyű legyen meglévő grafikus csővezetékekbe integrálni.

Bemutatunk egy gyors, robosztus módszert a valós idejű puha szélű árnyékok renderelésére.

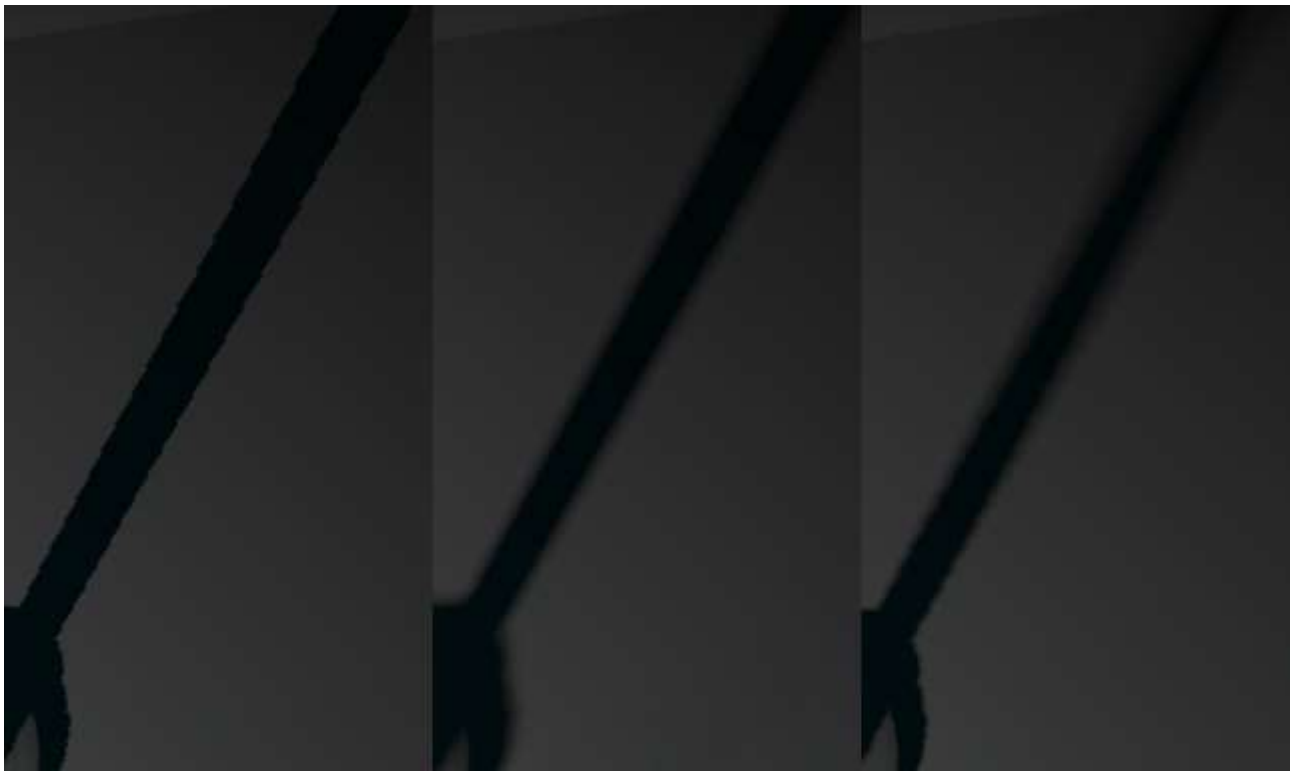
2. TÖRTÉNELEM

A valós idejű képszintézis területén általában olyan világítási technikákat használnak, amik lokális illuminációs modelleken alapulnak, amik csak a felületre közvetlenül eső fényt veszik számításba (így egyszerűsítést alkalmaznak), és emiatt nem veszik figyelembe a környezet takarásának hatását (árnyékok), így ezt külön szimulálni kell. Ezzel ellentétben a globális illumináció alapú algoritmusoknál az árnyékok külön számítás nélkül létrejönnek. A következőkben a lokális illuminációt kiegészítő, valós idejű árnyék szimuláló algoritmusokról lesz szó. Az ilyen módszerek szükségképp nagyon hatékonyak kell legyenek, ugyanis valós idejű grafikában az egész jelenet kiszámítására körülbelül 40ms áll rendelkezésre.

Hagyományosan a kemény szélű árnyékokat kétféle technikával állítják elő: az árnyék vetítéssel [Williams78], illetve az árnyék térfogatokkal [Crow77]. Az árnyék vetítés úgy működik, hogy a jelenetet leképezzük a fényforrás nézőpontjából, eltároljuk a jelenet mélységet, azaz a kamerától való távolságot reprezentáló képet, és később a világítás során ezt a mélység puffert mintavételezzük. A mintákat áttranszformáljuk a néző koordináta-rendszerébe, és összehasonlítjuk a mélységgel, hogy megállapítsuk, hogy a jelenet egy bizonyos pontja árnyékban van-e.

Az árnyék térfogatok úgy működnek, hogy a teret speciális árnyék testekkel felosztjuk árnyékban levő és világos részekre. Annak meghatározása, hogy egy felületi pont árnyékban van-e, a nézeti sugaraknak az árnyéktestek oldallapjaival való metszéspontok számából meghatározható. Ennek a technikának nagy hátránya, hogy gyakran kitöltési-ráta érzékennyé válik, amely alacsony teljesítményhez vezet [Nealen02]. Emiatt az árnyékvetítést fogjuk használni.

Az árnyék térfogatokkal pixel-pontos kemény szélű árnyékokat lehet képezni, ezzel szemben az árnyék vetítés minősége attól függ, hogy mekkora az árnyék-térkép (mélység-puffer) mérete. Ha nem áll rendelkezésre elég minta egy ponton, akkor alul-mintavételezés fog fellépni, amely képi hibákhoz vezethet. Ha több minta áll rendelkezésre, mint szükséges lenne, akkor viszont túl-mintavételezés fog fellépni, ami elpazarolt memória sávszélesség-használatot eredményez. Az árnyékvetítést sújtják továbbá a vetítési képi hibák, a perspektivikus képi hibák, és a hibás ön-árnyékolás is, amelyek eliminálására megfelelő figyelmet kell szentelni.



Ábra 2 Kemény szélű árnyékok (balra), egyenletes lágyságú árnyékok PCF-el számítva (középen), SSSS-el számított változó lágyságú árnyék. Változó lágyság használatával az árnyékok élesebbek lesznek az árnyékoló objektumhoz közeledve

Ahhoz, hogy lágy szélű árnyékokat is meg tudjunk jeleníteni árnyék vetítéssel, azt gyakran kiegészítjük árnyék szűréssel. Erre jelent megoldást a percentage-closer filtering (PCF) nevű technika, melyet [Reeves87] vezetett be. Ez a technika a lágyszélű árnyékokat árnyék-térbeli elmosással éri el. Később a PCF-et kiegészítették egy képtérbeli elmosó lépéssel [Shastry05], ami még nagyobb kernelméretet tesz lehetővé. Azonban ezek a technikák csak egyenletes lágyságú árnyékokat tudnak leképezni. Az 2-es ábrán látható egy összehasonlítás a különböző lágyszélű árnyék számító algoritmusokról.

A percentage-closer soft shadows (PCSS) erre jelent megoldást, segítségével változó lágyságú árnyékokat lehet megjeleníteni [Fernando05]. A PCSS úgy működik, hogy a szűrés méretét változtatja a PCF elmosás részénél. Egy ún. blokkoló keresést végez annak érdekében, hogy megbecsülje a lágyság mértékét egy bizonyos pontban. A blokkoló keresés során tulajdonképpen egy nxn-es területen keressük azokat a pontokat, amelyek potenciálisan eltakarják a fényforrás nézőpontjából a vizsgált pontot. Ezeknek a blokkoló pontoknak a vizsgált ponttól vett távolságainak az átlaga lesz a keresett lágyság mértéke. A fényforrás méretével arányos a terület mérete, ahol a keresést kell végezni. Emiatt nagyobb fényforrásoknál több mintára is szükség lehet, mint általában.

Az árnyékok széleinek változó mértékű elmosásához a módszer a fenti lágyság értéket használja fel. Azonban a PCSS még mindig árnyék térben végzi az elmosást, emiatt a teljesítmény függeni fog az árnyéktérkép méretétől és a kernel méretétől. Ez teljesítményromláshoz vezethet, főleg, ha nagyszámú lámpát jelenítünk meg.

A screen-space soft shadows (SSSS) [Gumbau10] technika ezt próbálja megoldani azzal, hogy az elmosást elhalasztja egy képtérbeli számításig, hogy a teljesítmény független legyen az árnyéktérkép méretétől. Képtérbeli elmosásnál azonban figyelembe kell venni a változó betekintési szöget, és emiatt anizotropikus, nem szeparábilis kernelű szűrést kell alkalmazni. Mivel a blokkoló keresés ettől függetlenül egy drága lépés lesz ($O(n^2)$, ahol n a kereső kernel mérete, amit minden árnyéktérkép pixelen lefuttatunk), ezért az SSSS-t kiegészítették [Gumbau10] egy alternatív módszerrel a lágyság méret becslésre. Ez a módszer egy minimum-szűrést alkalmaz az árnyéktérképre. Továbbá a szűrés szétválasztható, és az eredmény elég ha durva közelítés (mivel ezt nem közvetlenül árnyék számításra fogjuk használni, hanem csak a lágyság mértékének becslésére), emiatt egy alacsony felbontású eredmény is elfogadható ($O(n + n)$, sokkal kisebb n-re). [Engel10] továbbá kiegészítette ezt az exponenciális árnyéktérkép technikával és egy javított távolság függvénnyel. Így az eredmény önárnyékolásmentes, és egyéb képi hibáktól mentes, továbbá jobban kihasználja ugyanazt a kernel méretet ha távolról nézi a szemlélő a jelenetet.

A Mipmapped Screen Space Soft Shadows (MSSSS) [Aguado és Montiel 11] technika tovább próbálja javítani a teljesítményt.

Ábra 3 Átlapolódó árnyékok hibás kezelése képi hibákhoz vezet (balra), átlapolódó árnyékok helyes megjelenítése

Az árnyéktérkép elmosási problémát áttranszformálja egy mipmap szint kiválasztási problémává az alapján, hogy mekkora lágyság kívánatos. Továbbá kiegészíti ezt a technikát, hogy figyelembe vegye a többszörös árnyékokat több árnyéktérkép használatával.

Ezek a képtérbeli technikák hasznosak, mivel állandó a szűrés költsége, tehát nem kell külön-külön minden fényforrás árnyéktérképét szűrni, elegendő csak a képtérben egyszer.

Emiatt a világítás és az árnyékolás függetlenek, és ez hatalmas kernel méretet tesz lehetővé, ami olcsó ám nagy lágyságot tesz lehetővé. Azonban a képtérben még egy probléma felmerül: több fényforrás átlapolódó árnyékai. Erre kínálunk megoldást a rétegelt árnyéktérképekkel.

Az ok, amiért szükségünk van rétegekre az az, hogy ha csak átlagoljuk a lágyság méreteket

(tehát csak egy lágyság méretet tárolunk pixelenként), akkor rossz értékeket és képi hibákat kapunk (ld. 3-as ábra), tehát az árnyékok nem néznek ki többé valósághűnek. Rétegekkel viszont el tudunk tárolni több lágyság méretet és árnyék adatot pixelenként, és emiatt az árnyékokat megfelelő mértékben el tudjuk mosni lágyságtól függően, akkor is, ha átlapolódnak egymásra.

Továbbá, a való életben az árnyékokat gyakran áttetsző objektumok vetítik így figyelembe kell vennünk az áttetsző árnyékokat is. [Aguado és Montiel 11] is figyelembe vette ezt a problémát, bár – a dolgozatban bemutatott módszerrel ellentétben – ott nem volt szó színes árnyékokról, csak áttetszőkről. Külön figyelmet fordítottak arra, hogy kiküszöböljék az egymást takaró blokkoló objektumokból eredő képi hibákat is.

A célunk az, hogy gyors, valósághű lágy szélű és akár színes árnyékokat jelenítsünk meg, változó lágysággal, figyelembe véve az átlapolódó és áttetsző árnyékokat is.

A 3-as ábrán látható a különbség az átlapolódó árnyékok helyes és helytelen kezelése között.

3. ALGORITMUS ÁTTEKINTÉS

3.1 G-buffer lépés

A G-buffer egy manapság elég népszerű kelléke a modern játékok grafikus csővezetékének. Ezt a buffert arra szokták használni, hogy a világítást késleltetve függetlenné tegyék azt a jelenet leképezésétől, így leegyszerűsítve azt. Ezt úgy érik el, hogy megtöltik ezt a képtérbeli puffert a világításhoz szükséges paraméterekkel, és így a világítás tulajdonképpen egy utóeffektusként tud működni. Továbbá ez a technika abban is segít, hogy a néző által látható pixeleken ne kelljen többször is kiszámolni a világítás értékét, mivel lehet, hogy az adott objektum a végső képen már takarásban lesz. A rajzolás komplexitása így a potenciálisan látható pixelek száma helyett (mely a képernyőfelbontáshoz képest több nagyságrenddel is nagyobb lehet) a képernyő felbontással lesz arányos – és természetesen a fényforrások számával is.

Ez a lépés eléggé algoritmus függő, és bár az eredeti SSSS algoritmusban ez benne volt az ún. távolsági térkép lépésben (l. később részletesen) mi úgy döntöttünk, hogy külön vesszük, hogy biztosan könnyen lehessen integrálni a technikát egy meglévő grafikus csővezetékbe. Lehetséges bármilyen G-buffer kialakítást használni, feltéve, hogy található benne legalább egy, a kamerától való távolságokat tartalmazó mélységpuffer, illetve egy, a pixelben látható felületi pontokban lévő felületi normálvektorokat tároló normál-puffer, mivel ezekre később szükség lesz. Fontos megjegyezni, hogy nem számít, hogy deferred shading, deferred lighting vagy bármilyen más népszerű technika van használatban.

Mi a deferred shading mellett döntöttünk az egyszerűsége és kiváló sebessége miatt.

A mi G-buffer kialakításunk a következő:

- D24 mélység-puffer (a néző és a jelenet egy pontja közötti távolságot tárolja)
- RGBA8 kamera-térbeli normál vektorok (az RGB csatornákon, az alfa szabad)

4. ÁRNYÉK TÉRKÉP KÉSZÍTÉS LÉPÉS

Ebben a lépésben a szokásos árnyék térkép készítést végezzük el, leképezve a jelenetet a fényforrás nézőpontjából és lementve az így kapott mélységpuffert.

Ez a technika jelenleg reflektorszerű, pontszerű, illetve nap-szerű (irány) fényforrásokat támogat. Mivel lágy árnyékokat jelenítünk meg, ezért a fényforrásoknak bár továbbra sem lesz térfogata (a világítás nem változik), a végső képen az árnyékok miatt úgy fog kinézni, mintha lenne.

A pontszerű fényforrásokat hat különböző, reflektorszerű fényforrásként kezeljük. Ez azt jelenti, hogy hatékonyan ki tudjuk választani a nem látható reflektorszerű fényforrásokat, akár a pontszerű fényforrások egy részét is (amiknek szintén el kéne készíteni az árnyék-térképét), amik nem fogják befolyásolni a végső képet. Ezen kívül érdemes külön figyelmet szentelni a vetítési, perspektivikus és ön-árnyékolási képi hibák elkerülésére.

Ezt a lépést kiegészítjük azzal, hogy figyelembe vesszük azt a szituációt, amikor az árnyék képző objektumok áttetszőek. Ebben a helyzetben némi fény keresztüljut az árnyék képzőn és az árnyék akár színes is lehet. Emiatt az árnyéktérképekben nem csak a jelenet mélységét, hanem az árnyék színét is el kell menteni. Az 4-es ábrán látható az ilyen színes árnyéktérkép tartalma.

Ábra 4 A színes árnyéktérkép tartalma: egy piros pózna megjelenítve a fényforrás nézőpontjából

5. RÉTEGEK BEVEZETÉSE ÉS FÉNYEK BEOSZTÁSA

Ahhoz, hogy több fény átlapolódását is helyesen meg tudjuk jeleníteni az eredeti SSSS technikával, azt is meg lehetne csinálni, hogy az eredeti technikát többször elvégezzük minden egyes fényforrásra, azonban ez elég költséges művelet lenne. Emiatt egy alternatív módszert kerestünk, amivel képtérbeli anizotropikus elmosást tudnánk végezni minden egyes átlapolódó árnyékra egy lépésben.

A kutatás során kialakult egy kulcs gondolat (amelyet később [Anichini14] is leírt): ha két fényforrás által megvilágított terület nem keresztezi egymást, akkor az árnyékaik sem fogják.

Ez azt jelenti, hogy képtérben a fényforrások árnyéka sem fog átlapolódni, és emiatt azon fényforrások árnyékai, melyek nem keresztezik egymást, egy lépésben elmoshatóak. Azonban a több fényforrástól származó, átlapolódó árnyékokat is figyelembe kell venni.

Megfigyeléseink során észrevettük, hogy ez a probléma (átlapolódó árnyékok elmosása) tulajdonképpen egy gráf színezési probléma, ahol minden egyes szín tulajdonképpen egy árnyék réteget jelképez.

Minden egyes árnyék rétegben több fényforrás árnyékainak az adatait tároljuk, melyek nem lapolódnak át. Ebben a gráfban minden fényforrás egy csúcs lesz, és akkor lesz köztük él, ha két fényforrás keresztezi egymást. Mivel a gráf színezéshez szükséges színek számának megtalálása általában NP teljes probléma, ezért úgy döntöttünk, hogy egy egyszerű mohó algoritmust fogunk használni, némi heurisztikával kiegészítve.

Például, mivel a nap várhatóan minden felületet megvilágít, ezért annak egy saját réteget biztosítunk.

A gráf színezését minden kiszámolt képkockánál el kell végezni, hogy a fényforrások dinamikusak lehessenek.

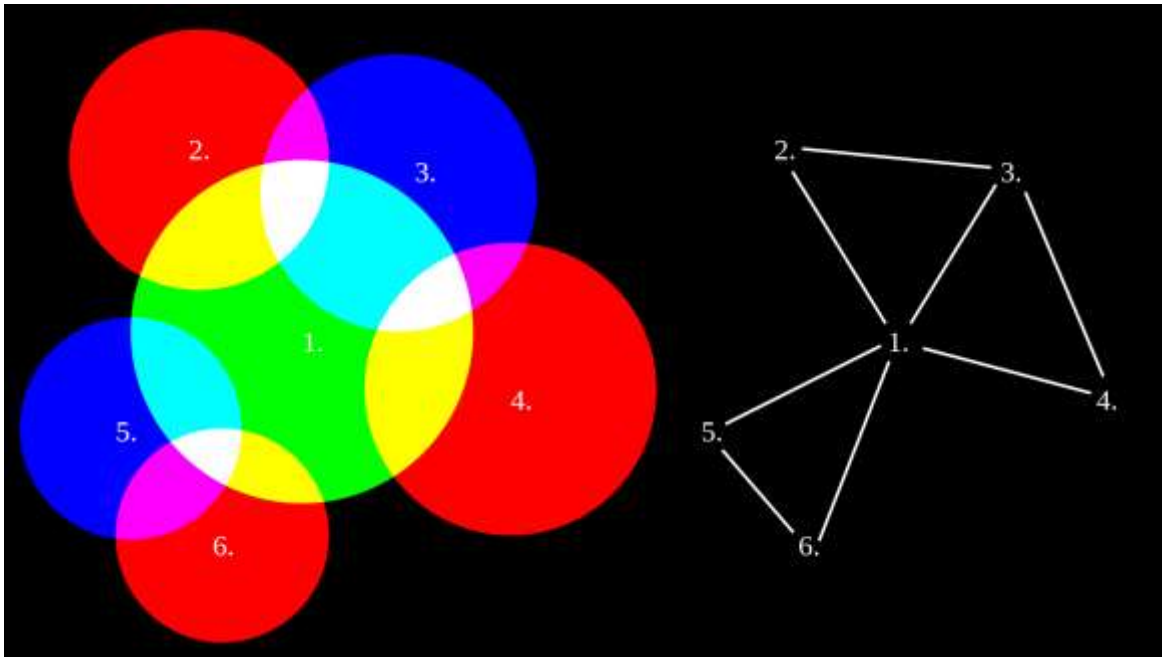
A mohó algoritmus használata azt jelenti, hogy a szükséges színek száma egyenlő lesz a maximum fokszám plusz eggyel. Ezért képtérbeli rétegek száma, melyeket szűrni kell, csak az átlapolódó árnyékok számának maximumától függ (maximum fokszám).

A mohó színezés egy hátránya, hogy a minősége nagyban függ a sorrendtől, amiben vesszük a csúcsokat. Emiatt egy gyakran használt csúcs rendezési módszert vetünk be, mégpedig azt, hogy a csúcsokat a fokszámuk alapján növekvő sorrendben vesszük.

Tulajdonképpen amit a rétegekkel csinálunk az az, hogy minimalizáljuk a memória és sávszélesség igényét az átlapolódó árnyékok szűrésének azzal, hogy rétegekbe csoportosítjuk az árnyékokat ahelyett, hogy egyenként tárolnánk és szűrnénk őket.

Továbbá tanácsos a rétegek számát előre lekorlátozni egy-egy jelenethez, így az általános színezési problémát egy k-színezési problémává lehet transzformálni (ahol k az előre definiált korlát a rétegek számára vonatkozóan). Az árnyék szűrésnek így előre becsülhető maximum költsége lesz, és a művészeknek meg lehet mondani, hogy ne használjanak több átlapolódó árnyékot, mint amennyit a korlát enged.

Ábra 5 A pontszerű fényforrások a hozzájuk tartozó rétegek szerint vannak színezve. Mindegyik réteget egy színnel jeleztünk (piros, zöld, kék és sárga). A fehér kockák a fényforrások pozícióját jelzik.



Ábra 6 A fényforrásokat megszámoztuk, és körökkel jelenítettük meg őket (balra). Mindegyik szín egy réteget jelképez (piros, zöld és kék). A fényforrások és egymással való átlapolódásukat a jobboldali ábrán egy gráfon ábrázoltuk. Látható, hogy az első lámpának a fokszáma 5, tehát maximum 6 rétegre lenne szükség ahhoz, hogy megjelenítsük ezeket a fényforrásokat. Azonban ebben az esetben egy jó színezés használatával a szükséges rétegek számát lecsökkenthetjük 3-ra.

Továbbá a fényforrás keresztezések kiszámolásának felgyorsítására használhatunk bármilyen

gyorsítóstruktúrát, mint például egy octreet. A konkrét réteg kialakítás a használt technikától függ (l. később).

Az 5-ös ábrán láthatóak az árnyék rétegek, az 6-on pedig a gráf.

6. RÉTEGEZETT LÁGYSÁG TÉRKÉP LÉPÉS

Ebben a lépésben minden rétegre kiszámítjuk az anizotropikus Gauss elmosáshoz szükséges paramétereket, többnyire úgy, ahogy a [Gumbau10]-ben le van írva. Ezek a következőek:

- a szemlélő és a jelenet egy megjelenített pontja közötti távolság (jelenet mélysége)
- az árnyék képző objektum és a jelenet egy megjelenített pontja közötti távolság (lágyság mérete)
- lágyság maszk, ami meghatározza, hogy egy pontot el kell-e mosni
- árnyék-térkép (bináris érték) vagy exponenciális árnyék-térkép (lebegőpontos számítás, opcionális)
- átlátszóság-térkép (RGBA érték)

6.1 Jelenet mélység

Mivel ezt az értéket már kiszámítottuk a G-buffer lépésnél, ezért nem szükséges megint kiszámolni.

6.2 Rétegezett lágyság-térkép

Kétféle módon lehet kiszámolni a lágyság méretét, ahogy [Gumbau10] is leírta:

- blokkoló kereső konvolúciót használva
- szétválasztható minimum szűrést használva (l. később)

Az elsőt, a blokkoló keresést [Fernando05] úgy kell elvégezni, hogy egy konvolúciót végzünk az árnyék térképen, potenciális blokkolókat keresve (tehát ahol egy pont árnyékban van), aztán ezeknek a blokkolóknak a fényforrástól vett távolságainak az átlagát felhasználva (a fényforrás projekciós terében véve) megbecsüljük a lágyság méretét egy adott pontban. A becslést a következő egyenlettel lehet elvégezni:

$$w_{penumbra} = \frac{\left(d_{receiver} - d_{blocker}\right)}{d_{blocker}} \cdot w_{light}$$

Egyenlet 1

ahol w_{light} a fényforrás mérete, amit egy grafikusnak kell majd beállítania (empirikus érték), $d_{observer}$ a szemlélőtől vett távolság a fényforrás projekciós terében véve, $d_{blocker}$ az átlagos blokkoló távolság, $d_{receiver}$ pedig az adott pont távolsága (fény projekciós térben véve). A fényforrás projekciós terében vett mélységet elő tudjuk állítani egy sima mélység-térképből a szokásos OpenGL projekciós mátrix segítségével.

Tanácsos a lágyság értékeket egy legalább 16 bites lebegőpontos textúrában tárolni, ellenkező esetben képi hibák jelenhetnek meg (sávosodás). Ez azt jelenti, hogy ha négy réteget akarunk tárolni, akkor azt megtehetjük egy RGBA16F textúra segítségével. Mindegyik réteget egy színcsatornában tároljuk, így a grafikus hardver vektorműveleteit kihasználva a négy réteg egy időben, párhuzamosan dolgozható fel.

$$\begin{pmatrix} \frac{n}{r} & 0 & 0 & 0 \\ 0 & \frac{n}{t} & 0 & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

Ábra 7 A (szimmetrikus perspektivikus) OpenGL projekciós mátrix, ahol n a kamera közeli síkjától való távolság, f a távoli, t = n x tan(fov x 0.5), és r = aspect x t

```
//# of bits in depth texture per pixel
unsigned bits = 16;
unsigned precision_scaler = pow(2, bits) - 1;
//generates a perspective projection matrix
mat4 projmat = perspective(radians(fov), aspect, near, far);
//arbitrary position in view space
vec4 vs_pos = vec4(0, 0, 2.5, 1);
//clip space position
vec4 cs_pos = projmat * vs_pos;
//perspective divide
vec4 ndc_pos = cs_pos / cs_pos.w;
float zranged = ndc_pos.z * 0.5f + 0.5f; //range: [0...1]
//this goes into the depth buffer
unsigned z_value = floor(precision_scaler * zranged);

//helper variables to convert back to view space
float A = -(far + near) / (far - near);
float B = -2 * far * near / (far - near);

//get depth from the depth texture, range: [0...1]
float depth = texture(depth_tex, texcoord).x;
float zndc = depth * 2 - 1; //range: [-1...1]
//reconstructed view space z
float vs_zrecon = -B / (zndc + A);
//reconstructed clip space z
float cs_zrecon = zndc * -vs_zrecon;
```

Kód részlet 1. A fényforrás projekciós terében vett mélység előállítása a szokásos mélység térképből

Két lehetőségünk van arra, hogy a lágyság információt előállítsuk sok fényforrás esetén:

- egyenként előállíthatjuk őket, és additívan egymásra keverhetjük (blendelhetjük) őket
- kötegelhetjük őket, és egyszerre előállíthatjuk őket (l. később)

Amikor minden egyes fényforrás lágyság információit külön lépésben állítjuk elő, akkor minden pixelnél több réteg is lesz a lágyságtérképben, emiatt a lágyság információt minden réteghez külön kell tárolni. Ezt additív hardveres blendeléssel lehet elérni.

6.3 Rétegezett lágyság maszk

A lágyság maszk tulajdonképpen azt tárolja, hogy találtunk-e blokkoló objektumot a blokkoló keresés során. Észrevehetjük, hogy a lágyságméret amit korábban kiszámítottunk, az tulajdonképpen csak egy érték, ami megmondja, hogy mennyire kell elmosni az árnyékokat egy bizonyos ponton. Más szavakkal ez az érték fogja meghatározni az anizotropikus Gauss szűrés effektív méretét. Tehát a lágyság maszkot el tudjuk tárolni a lágyság méretben azzal, hogy nullára állítjuk, ha a maszk is nulla lenne. Így csak ellenőrizni kell, amikor az anizotropikus szűrést végezzük, hogy a lágyság méret nagyobb-e nullánál, és csak abban az esetben végezni elmosást.

6.4 Rétegezett árnyék-térkép

Két lehetőség van arra, hogy mit tároljunk az árnyék térképben:

- egy bináris értéket, ami azt mondja meg, hogy egy pont árnyékban van-e
- egy exponenciális árnyéktérkép értéket, ahogy [Engel10]-ben is le van írva, ami azt jelenti, hogy a bináris átmenetet egy exponenciális függvénnyel közelítjük, így elkerülve az önárnyékolást

Ha egy bináris értéket tárolunk, akkor a szokásos árnyék-tesztet az árnyék térkép mintavételezésével, a jelenet mélység átvetítésével és a kettő összehasonlításával el tudjuk végezni, majd tárolni. Így egy árnyék réteget egyetlen biten el tudunk tárolni, tehát egy R32U (32 bites előjel nélküli egész típusú) textúrában 32 réteget tudunk tárolni.

Ha pedig az exponenciális árnyéktérkép értékkel akarunk dolgozni, akkor az eredményt legalább 8 biten kell eltárolni, hogy négy réteg beleférjen egy RGBA8 textúrába. Az érték, amit tárolni kell pedig a következő

$$tex(z,d,k) = e^{k\cdot(z-d)}$$

Egyenlet 2

ahol *z* a jelenet mélysége a szemlélő nézőpontjából, *d* a jelenet mélysége a fényforrás nézőpontjából, *k* pedig egy empirikus érték (skálázó tényező), amit az exponenciális árnyék térkép állítására használunk.

6.5 Rétegezett átlátszóság-térkép

Ahhoz, hogy az átlátszó objektumok által képzett árnyékoknál a lágy széleket meg tudjuk jeleníteni, el kell mosni az átlátszóságtérképet is az árnyék leképezés során.

Emiatt ezeket a térképeket egy rétegezett térképbe gyűjtjük. Minden réteget egy RGBA8 értékkel tudunk leírni, amit egy R32F lebegőpontos értékbe bele tudunk rakni. Ezért egy RGBA32F textúrában, aminek 128 bitje van 4 réteget tudunk tárolni. A tárolást elvégző kód a 2-es kód részletben látható.

```
//input: float value in range [0...1]
uint float_to_r8( float val )
{
  const uint bits = 8;
  uint precision_scaler = uint(pow( uint(2), bits )) - uint(1);
  return uint(floor( precision_scaler * val ));
}

uint rgba8_to_uint( vec4 val )
{
  uint res  = float_to_r8(val.x) << 24;
       res |= float_to_r8(val.y) << 16;
       res |= float_to_r8(val.z) << 8;
       res |= float_to_r8(val.w) << 0;
  return res;
}
```

Kód részlet 2. Egy függvény, ami egy RGBA8 értéket egy R32F lebegőpontos értékbe csomagol.

7. ANIZOTROPIKUS GAUSS ELMOSÁS LÉPÉS

Most pedig el kell mosni az árnyékokat, hogy változó lágyságú árnyékokat képezzünk. Az előző lépésben kiszámított lágyság információt fogjuk mintavételezni ehhez.

Hogy figyelembe vegyük a szemlélő lehetséges változó betekintési szögeit, módosítani kell a szűrő méretét. Ehhez a szűrés méretét közelítjük úgy, hogy a Gauss szűrő kernelt egy ellipszoidba vetítjük a geometria elhelyezkedését figyelembe véve. A [Geusebroek és Smeulders03]-ban használt technikát használtuk, amelyik bemutatja, hogy hogyan lehet anizotropikus Gauss szűrést végezni úgy, hogy a kernel szétválasztható maradjon. Továbbá figyelembe kell venni, ha az árnyékokat távolról nézzük, akkor a szűrés méretét csökkenteni kell, hogy az effektív szűrés mérete konstans maradjon. A skalárszorzat természete miatt azonban a szűrés mérete nagyon kicsire csökkenhet amikor alacsony szögből nézünk egy felületet, emiatt limitálni kell a szűrő minimális méretét. A minimum értéket empirikusan választjuk, 0.25 általában elég jól működik. A 3-as kódrészlet mutatja, hogy hogyan kell a változó szűrő méretet implementálni.

```
float threshold = 0.25;
float filter_size =
  //account for light size (affects penumbra size)
  light_size *
  //anisotropic term, varies with viewing angle
  //added threshold to account for diminishing filter size
  //at grazing angles
  sqrt( max( dot( vec3( 0, 0, 1 ), normal ), threshold ) ) *
  //distance correction term, so that the filter size
  //remains constant no matter where we view the shadow from
  ( 1 / ( depth ) );
```

Kód részlet 3. Változó szűrő méret implementációja

Ezután már csak a Gauss szűrőt kell kiértékelni. Mivel ez szétválasztható, ezért az elmosást két lépésre osztjuk, egy horizontálisra és egy vertikálisra. A szűrő méretét az anizotropikus értékkel fogjuk módosítani. Továbbá minden réteget egyszer mintavételezni kell minden iterációban és kicsomagolni az egyes rétegeket. A réteg kicsomagolást a 4-es kódrészlet mutatja be. Mivel ez egy képtérbeli szűrő, ezért figyelembe kell venni, hogy az árnyékokból akár szivároghat a fény, ha túl nagy a szűrőméret, ezt látni is lehet a 8-as ábrán. Ezért ha a mélységkülönbség a szűrő középpontja és a mintavételezett pont között nagyobb, mint a határérték (általában 0.03 jól működik), akkor nem vesszük figyelembe a mintát.

```
float unpack_shadow( vec4 shadow, int layer )
{
  //4 layers
  uint layered_shadow = uint(16.0 * shadow.x);
  return ( ( layered_shadow & ( 1 << layer ) ) >> layer );
}

vec4 hard_shadow = texture( layered_shadow_tex, texcoord );
float layer0 = unpack_shadow( hard_shadow, 0 );
```

Kód részlet 4. Árnyék adatok kicsomagolása

Mivel a Gauss szűrő költsége nagyban függ a pixelek számától ahol futtatni kell, ezért ha fél felbontáson futtatjuk, azzal sokat lehet spórolni a szűrés költségén. Ez nem eredményez a végső képen látható hibát, mivel az eredmény alacsony frekvenciás részletekből áll, elmosott lesz. Tehát, ha úgy döntünk, hogy a Gauss elmosást fél felbontáson akarjuk végezni, akkor

kihasználhatjuk azt, hogy az eredeti kemény árnyéktérkép még megvan az eredeti felbontásban, emiatt ha a kemény árnyéktérképet rács mintában mintavételezzük (négy mintával, a textúra gather műveletet felhasználva), akkor kiküszöbölhetünk néhány képi hibát (tulajdonképpen túlmintavételezve a rétegezett árnyék térképet). Úgy is dönthetünk, hogy exponenciális árnyéktérképeket használunk, ami csökkenteni fogja az ön-árnyékolásból adódó képi hibákat.

Ha vannak áttetsző árnyékképző objektumok, akkor azokat is el kell mosni. Ugyanúgy kell eljárni, mint a nem áttetszőeknél, kivéve, hogy a kicsomagolás más lesz. A kicsomagolást az 5-ös kódrészlet mutatja be.

```
vec4 uint_to_rgba8( uint val )
{
  uint tmp = val;
  uint r = (tmp & 0xff000000) >> 24;
  uint g = (tmp & 0x00ff0000) >> 16;
  uint b = (tmp & 0x0000ff00) >> 8;
  uint a = (tmp & 0x000000ff) >> 0;
  return vec4( r / 255.0, g / 255.0, b / 255.0, a / 255.0 );
}
```

Kód részlet 5. Egy függvény, ami egy RGBA8 értéket egy R32F lebegőpontos értékből kicsomagol.

Ábra 8 A képtérbeli szűrésből eredő képhibák

8. VILÁGÍTÁS LÉPÉS

Az utolsó lépésben pedig meg kell világítani a jelenetet. Ez a lépés nagyon implementáció függő, de ennek ellenére az SSSS-t könnyű integrálni egy világítási technikába. Mi egy optimalizált tiled deferred shadert használunk. Ez tulajdonképpen a deferred shading algoritmus optimalizált változata, ahol a képet felosztjuk egy alacsony felbontású rácsra (tile), és a fényforrásokat ezekbe gyűjtjük. Így az egyes fényforrások világítását a teljes kép helyett csak ott kell kiszámítani, ahol a rácsmezőket érintik.

A világításnál ki kell számítani, hogy a fényforrás melyik réteghez tartozik, és annak a rétegnek az elmosott árnyék-térképét kell mintavételezni, amivel a világítás eredményét beszorozzuk. Fontos megjegyezni, hogy elegendő mindössze egyszer mintavételezni az elmosott árnyék-térképet pixelenként, és csak felhasználni a megfelelő réteget.

9. TELJESÍTMÉNNYEL KAPCSOLATOS MEGFONTOLÁSOK

Több módja is van annak, hogy felgyorsítsuk az SSSS-el kapcsolatos számításokat. Például az anizotropikus Gauss elmosást lehet alacsonyabb (általában fél) felbontáson is végezni. Ezzel tulajdonképpen az elmosó algoritmus memóriasávszélesség-igényét csökkentjük.

Továbbá a lágyságinformáció számítását is végezhetjük alacsonyabb felbontáson, azonban ez néhány képi hibát is eredményezhet az árnyékképző objektumoknál (ahol a lágyság mérete kicsi), amit el lehet tüntetni a túlmintavételezéses módszerrel, amit fentebb leírtunk.

A minimum-szűrés módszert is implementálhatjuk. Ennél egy alacsony felbontású minimum-szűrt árnyéktérképet kell kiszámítani, az árnyéktérkép számítás lépés után. Ezzel az értékkel lehet helyettesíteni a blokkolókeresés eredményét. Továbbá a lágyság maszkot implementálhatjuk úgy is, hogy megnézzük, hogy a minimum-filterezés eredménye egy bizonyos határérték alatt van-e (ahol az anizotropikus Gauss szűrés effektív mérete amúgy is elhanyagolható lenne). Így nem kell eltárolni a lágyság méreteket külön képtérbeli rétegekben, mivel a minimum-szűrt térképek elég kicsik ahhoz, hogy mindegyiket egyesével mintavételezzük az anizotropikus elmosás során. Ilyenkor a lágyság térkép lépésben csak a rétegezett árnyéktérképet és a rétegezett átlátszóság térképet számítjuk ki. Az algoritmus többi része ugyanaz marad. Azonban azt mértük le 4 réteg és 16 fényforrás mellett, hogy ez a megközelítés nagyjából 0.2 ms-al lassabb, mint ha a szokásos módszerrel tárolnánk a lágyság méretet (rétegesen). Bár ez nem túl nagy különbség, a valós idejű algoritmusoknál általában minden apróság számít.

Lehet, azonban, hogy ez az eredmény több réteg esetén változna.

Továbbá, ahelyett hogy a lágyság információt minden fényforrásra külön kiszámoljuk és összeblendeljük őket additívan, akár azt is lehet csinálni, hogy a fényforrásokat kötegeljük, és a lágyság információt egy lépésben kiszámítjuk. Azonban, ha ezt az utat választjuk, akkor az összes árnyéktérkép és minimum-szűrt árnyéktérkép információnak egyszerre rendelkezésre

kell állnia. Ez azt jelenti, hogy egy lépésben az összes fényforrás összes árnyéktérképét mintavételezni kell. Ezeket át lehet adni egy árnyalóprogramnak textúra tömbök vagy atlaszok használatával.

A Gauss elmosás kernel méretét is tudjuk állítani. Általában egy 11x11-es szűrő jól működik, de akár egy hatalmas 23x23 szűrőt is lehet integrálni, ami lehetővé teszi a hatalmas lágy szélű árnyékokat is. Vagy akár lehet egy olcsó, 5x5-ös szűrőt is használni.

10. EREDMÉNYEK

A teszteket egy PC-n futtattuk, amiben egy Core i5 4690 processzor, 8-GB DDR3 RAM és egy Radeon R9 270X 2-GB grafikus kártya volt. Tesztként egy textúrázatlan Sponza jelenetet használtunk, 16 színes fényforrással, mindegyiknek pedig egy 1024x1024-es árnyéktérképe volt, hogy illusztráljuk az átlapolódó árnyékokat.

11. MINŐSÉG TESZTEK

A referencia képeket a Blender sugárkövető rendszerével állítottuk elő, és ezekhez hasonlítottuk az eredményeinket, folyamatosan ellenőrizve, hogy megfelelő minőségű eredményt kapjunk. A 9-es ábrán látni lehet az összehasonlítást.

Ábra 9 Balra: árnyékok SSSS-el megjelenítve. Jobbra: reeferencia kép, Blenderrel készítve

Amint látni lehet, az eredményeink látszólag azonosak a referenciával, azonban a Gauss szűrő mérete hatással lehet a kimenetre. Mivel az elmosást képtérben végezzük, ezért könnyedén megengedhetünk magunknak hatalmas szűrő kernel méreteket is. Fontos megjegyezni, hogy a fényforrások méretét empirikusan választottuk, hogy minél közelebbi eredményt kapjunk a referenciához.

A 10-es ábrán látni lehet további példákat a Sponza jelenetből.

12. TELJESÍTMÉNY ANALÍZIS

Az 1-es táblázatban látni, hogy milyen teljesítményértékeket mértünk a Sponza jelenet megjelenítése során, 16 fényforrással, ugyanabból a nézőpontból a mi technikánkat használva, a PCSS, PCF és kemény szélű árnyékokat használva. A referenciaképen, a mi technikánkat használva, illetve a PCSS esetben az árnyékoknak változó lágyságú szélük van. A PCF verzióban egyenletes szélű lágy árnyékok vannak, és a kemény szélű árnyékoknál értelemszerűen nincs lágy szél.

Az algoritmusunk jó pár verzióját lemértük, úgy mint a min-szűréssel optimalizált változatot

és a blokkolókeresést használó változatot. Továbbá lemértünk egy általunk optimalizált változatot is, ami minimum-szűrést használ, fél felbontáson futtatja a Gauss elmosást, a lágyságinformáció számítást, illetve használja a fent említett túlmintavételezési technikát, hogy a minőség megmaradjon. Mindegyik változat kötegelést használ a rétegezett lágyság információ számításánál.

16 fényforrás, 4 réteg	**720p**	**1080p**
világítás	1.0ms	1.9ms
Kemény szélű árnyékok	12.5ms	13.9ms
PCF 5x5	14.9ms	18.8ms
PCF 11x11	21.3ms	31.0ms
PCF 23x23	40.0ms	59.4ms
PCSS 5x5 + 5x5	17.1ms	22.6ms
PCSS 5x5 + 11x11	21.5ms	31.9ms
PCSS 5x5 + 23x23	40.1ms	59.5ms
SSSS blokkoló keresés 5x5 + 5x5	24.3ms	33.0ms
SSSS blokkoló keresés 5x5 + 11x11	24.7ms	34.4ms
SSSS blokkoló keresés 5x5 + 23x23	25.7ms	37.1ms
SSSS minimum szűrő 5x5	16.5ms	19.2ms
SSSS minimum szűrő 11x11	16.9ms	20.3ms
SSSS minimum szűrő 23x23	17.8ms	22.7ms
SSSS optimalizálva 5x5	16.0ms	18.0ms
SSSS optimalizálva 11x11	16.5ms	19.1ms
SSSS optimalizálva 23x23	17.4ms	21.1ms

Táblázat 1 Teljesítmény mérések (teljes képkocka ideje) a Sponza jelenetből

Amint látható a mi módszerünk lényegesen jobban teljesít a PCSS-nél, és nagyjából ugyanannyiba kerül, mint a PCF. Megfigyelhető, hogy míg az árnyéktérbeli elmosáson alapuló technikák (PCF, PCSS) teljesítménye csökken amikor a felbontást növeljük, addig a mi módszerünk nem szenved ettől a problémától. Továbbá míg a Gauss elmosás kernel méretének növelése nem befolyásolja számottevően a mi módszerünk teljesítményét, addig az árnyéktérbeli elmosáson alapuló technikáknál ez jelentős teljesítményromláshoz vezet.

		1080p
Fényforrások száma	**PCSS 5x5 + 11x11**	**SSSS** optimalizálva **11x11**
16	31.9ms	19.1ms
12	28.1ms	15.4ms
8	24.0ms	11.8ms
4	18.1ms	7.9ms
1	10.2ms	4.1ms

Táblázat 2 A fényforrások számának változására való érzékenység mérése

A 2-es táblázatból az is kiderül, hogy a mi módszerünk sokkal jobban skálázódik a

fényforrások számának drasztikus emelkedésével. Az idő nagy részét így az árnyéktérkép kiszámítása teszi ki.

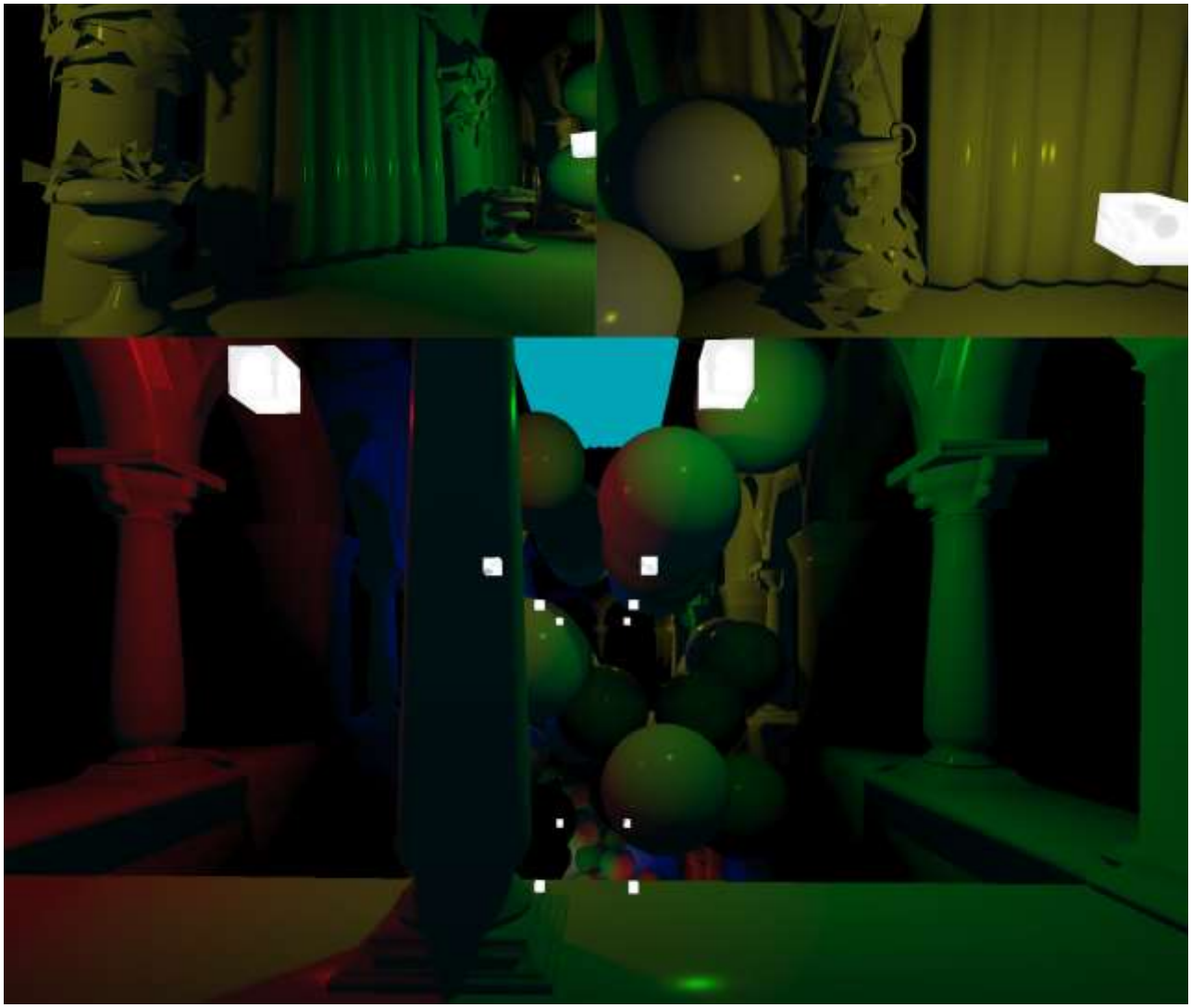
13. KONKLÚZIÓ

Megmutattuk, hogy rétegezett árnyék térképek használatával rendesen meg lehet jeleníteni az átlapolódó árnyékokat, és, hogy használhatunk rétegezett átlátszóság térképeket is, hogy áttetszőek lehessenek az árnyékképző objektumaink. Azt is megmutattuk, hogy ezt a technikát valós időben lehet implementálni úgy, hogy a minőség nem romlik. Továbbá láthattuk, hogy a fényforrások számának drasztikus emelkedésével a teljesítmény szemben a többi technikával alig romlik. Ugyanezt tapasztalhattuk a felbontás növekedésével, és a szűrő méretének változtatásával. Végül azt is láthattuk, hogy az áttetsző színes árnyékvetítő objektumokat is helyesen lehet ezzel a technikával kezelni.

14. JÖVŐBELI TERVEK

Az új grafikus API-kkal (DirectX12, Vulkan, stb.) a jövőben sokkal hatékonyabban lehet majd bizonyos algoritmusokat megvalósítani. Ez erre a technikára is igaz. A rétegezett lágyság információk számítását fel lehet gyorsítani azzal, hogy kihasználjuk a GPU-kban levő különböző erőforrások kihasználatlanságát, ami akkor jelentkezik amikor az árnyéktérképeket számítjuk. Így gyakorlatilag a számításokat időben átlapoljuk, és az árnyéktérkép-számításon felül felmerülő extra költségeket eliminálhatjuk.

Továbbá a lágyság maszk és lágyság méret számítását külön lehetne választani, mivel míg előbbi érzékenyebb a felbontására (főleg ha a lágyság mérete alacsony), az utóbbinál ez annyira nem fontos.

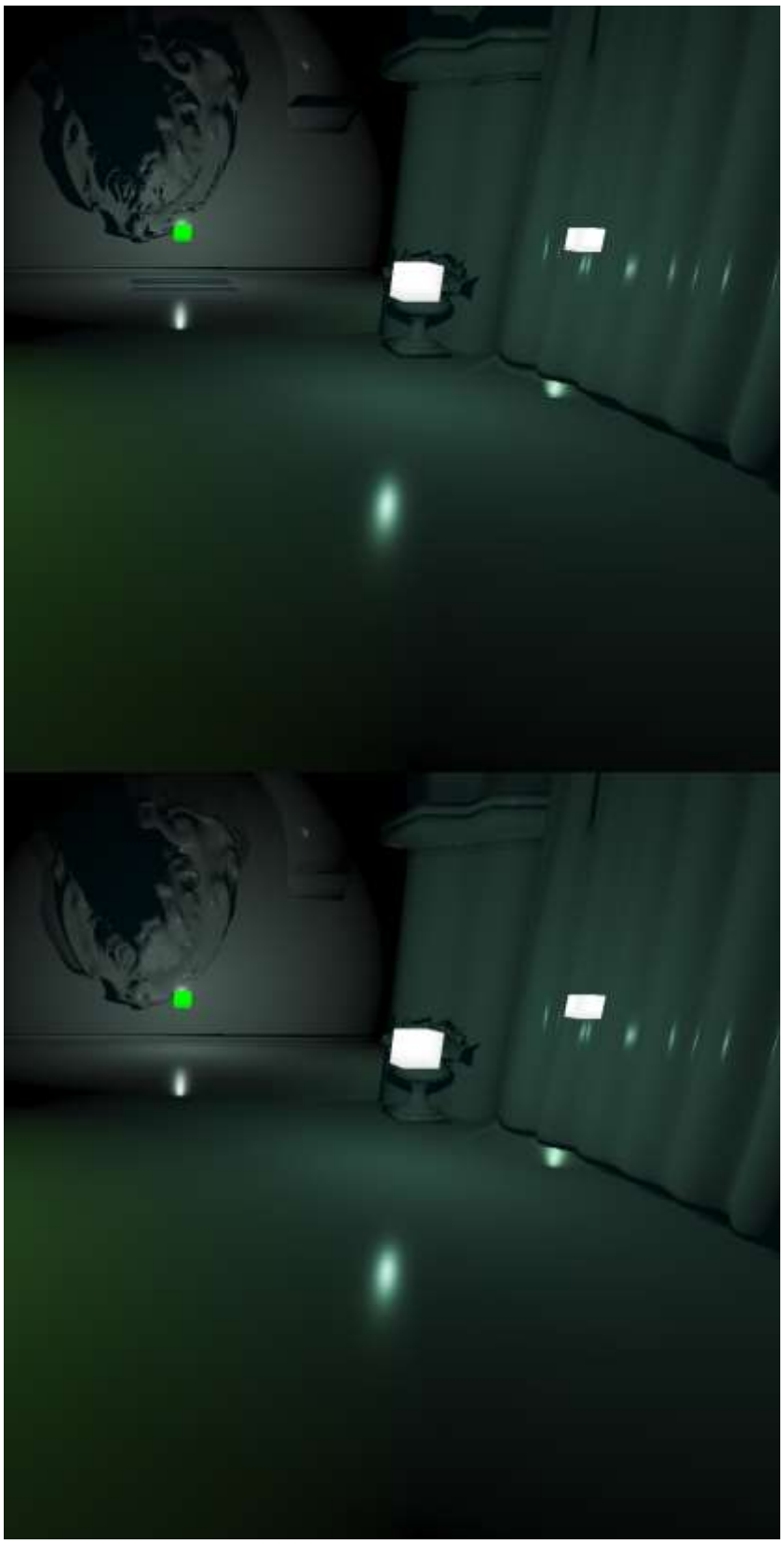


Ábra 10 SSSS-el készített képek, amin a kockák a fényforrásokat ábrázolják

Referenciák

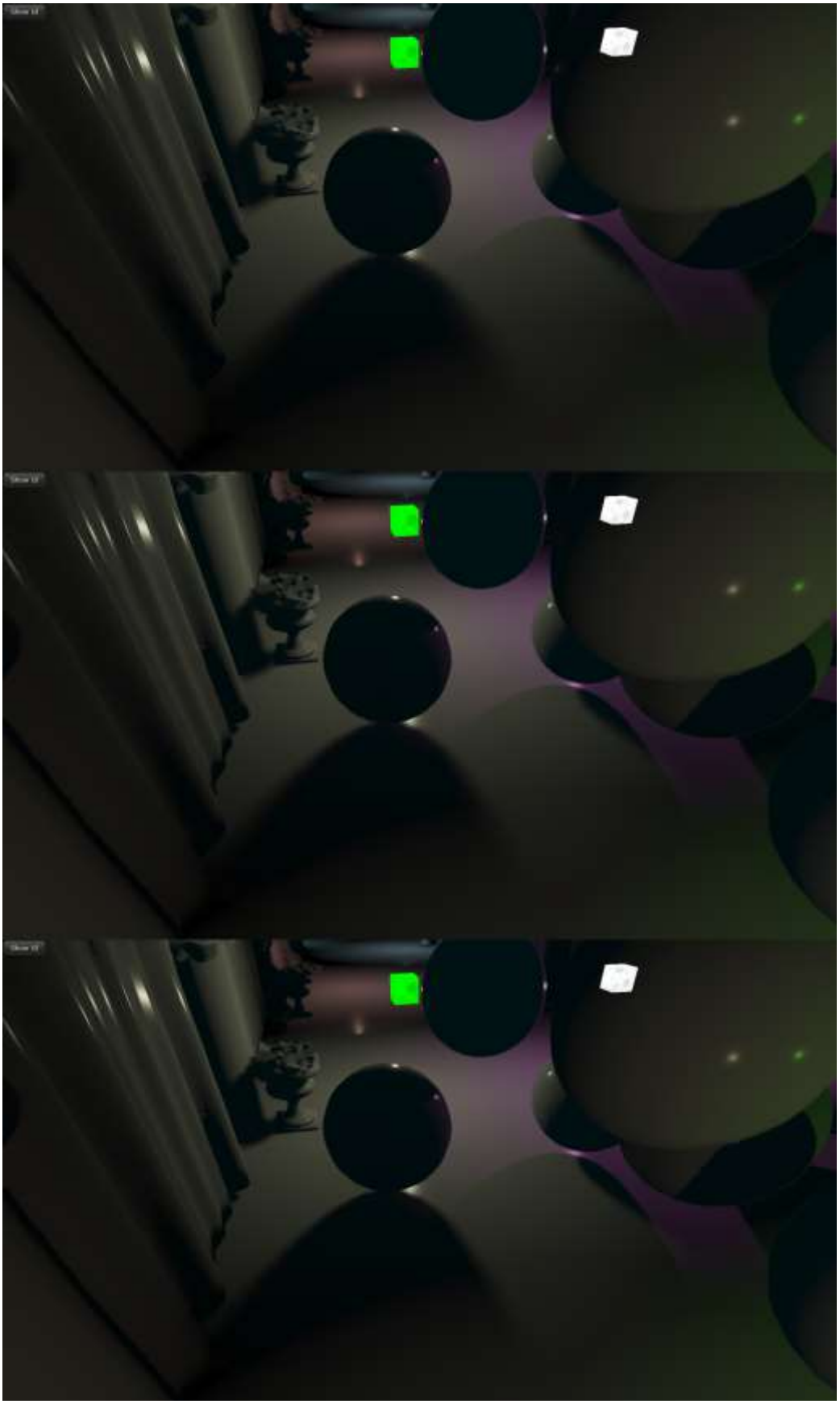
[Williams78] Lance Williams, “Casting Curved Shadows on Curved Surfaces”, SIGGRAPH '78, 12, 12, pp. 270-274, 1978
[Anichini14] Steve Anichini, “Bioshock Infinite lighting”, http://solid-angle.blogspot.hu/2014/03/bioshock-infinite-lighting.html, retrieved 25th July 2014
[Aguado11] Alberto Aguado and Eugenia Montiel, “Mipmapped Screen-Space Soft Shadows”, GPU Pro 2, 2011
[Engel10] Wolfgang Engel, “Massive Point Light Soft Shadows”, 2010
[Gumbau10] Jesus Gumbau, Miguel Chover and Mateu Sbert, “Screen Space Soft Shadows”, GPU Pro, 2010
[Fernando05] Randima Fernando, “Percentage-Closer Soft Shadows”, SIGGRAPH '05, ACM SIGGRAPH 2005 Sketches, p. 35, 2005
[Shastry05] Anirudh. S Shastry, “Soft-Edged Shadows”, http://www.gamedev.net/page/resources/_/technical/graphics-programming-and-theory/soft-edged-shadows-r2193, 2005, retrieved 25th July 2014
[Reeves87] William T. Reeves, David H. Salesin and Robert L. Cook, “Rendering Antialiased Shadows with Depth Maps”, 1987
[Nealen02] Andrew V. Nealen, “Shadow Mapping and Shadow Volumes: Recent Developments in Real-Time shadow Rendering”, 2002
[Crow77] Franklin C. Crow, "Shadow Algorithms for Computer Graphics", 19Computer Graphics (SIGGRAPH '77 Proceedings), vol. 11, no. 2, 242-248, 1977
[Geusebroek és Smeulders 03] Jan M. Geusebroek and Arnold W. M. Smeulders. “Fast Anisotropic Gauss Filtering” IEEE Transactions on Image Processing 12:8, pp 99-112, 2003

Függelék

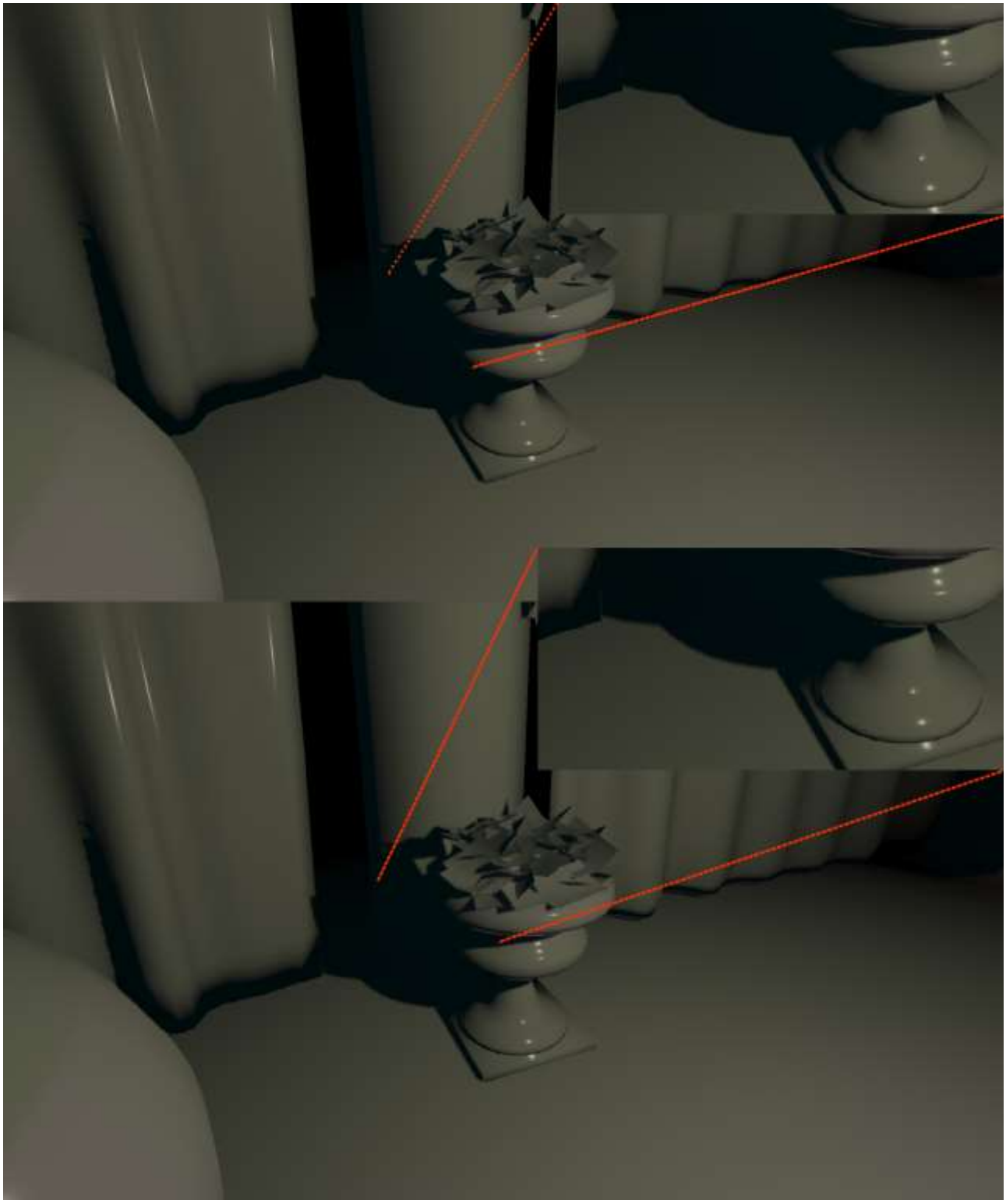


Ábra 11 Az exponenciális és sima árnyéktérképek közötti különbség. Megfigyelhető az önárnyékolás képi hiba.

Ábra 12 A minimum szűrés (alul) és blokkoló keresés (felül) algoritmusok közötti különbség



Ábra 13 A szűrőméretek közötti különbség. 23x23-as Gauss szűrő (felül), 11x11-es szűrő (középen), 5x5-ös szűrő (alul)



Ábra 14 A túlmintavételezés hatása. Látható, ahogy kisimítja a képi hibákat, ahol a Gauss szűrő effektív mérete alacsony.

Ábra 15 Több, átlapolódó fényforrás árnyékai

Ábra 16 Helyes önárnyékolás

Ábra 17 Példa 1

Ábra 18 Példa 2

Ábra 19 Példa 3