SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 2162

PRIKAZ SJENA KORIŠTENJEM MAPE SJENE

Mate Buljan

Zagreb, lipanj 2020.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 2162

PRIKAZ SJENA KORIŠTENJEM MAPE SJENE

Mate Buljan

Zagreb, lipanj 2020.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 13. ožujka 2020.

DIPLOMSKI ZADATAK br. 2162

Pristupnik:	**Mate Buljan (0036490294)**
Studij:	Računarstvo
Profil:	Računarska znanost
Mentor:	prof. dr. sc. Željka Mihajlović

Zadatak: **Prikaz sjena korištenjem mape sjene**

Opis zadatka:

Proučiti postupke ostvarivanja sjena, posebice korištenjem mape sjene te probleme i nedostatke pojedinih postupaka. Razraditi različite varijante postupaka ostvarivanja sjene metodom mape sjene koje ostvaruju poboljšanje kvaliteta rezultata. Razraditi ostvarivanje mekih sjena, koristiti filtriranja, zaglađivanja i strukture poput kaskadiranja. Načiniti programsku implementaciju koja omogućuje analizu i usporedbu razrađenih postupaka. Na različitim primjerima prikazati i usporediti ostvarene rezultate. Diskutirati utjecaj različitih parametara. Načiniti ocjenu rezultata i implementiranih algoritama. Izraditi odgovarajući programski proizvod. Koristiti programski jezik C++ i grafičko programsko sučelje DirectX. Rezultate rada načiniti dostupne putem Interneta. Radu priložiti algoritme, izvorne kodove i rezultate uz potrebna objašnjenja i dokumentaciju. Citirati korištenu literaturu i navesti dobivenu pomoć.

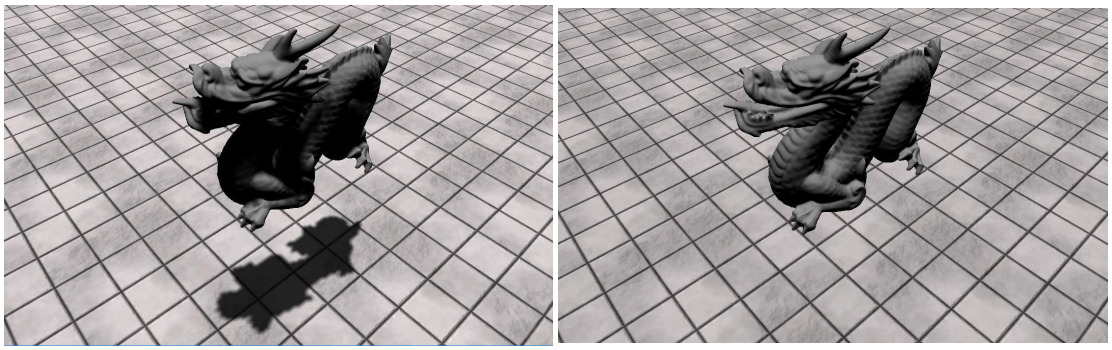
Rok za predaju rada: 30. lipnja 2020.

SADRŽAJ

1. Uvod

Sjene su vrlo važne za prikaz realističnih virtualnih scena. Osim toga, sjene pružaju informacije poput:

- Prostornog rasporeda objekata u sceni kao što se vidi na slici 1.1.
- Oblik objekta koji baca sjenu. Ako objekt trenutno nije vidljiv, a njegova sjena je, to prije svega pruža informaciju o postojanju i poziciji tog objekta.
- pozicije i oblika izvora svjetlosti.



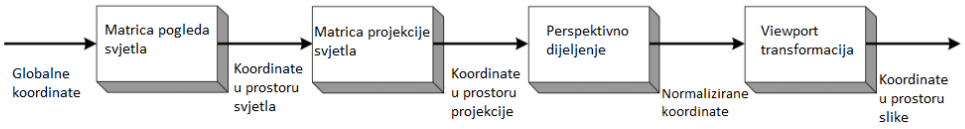
Slika 1.1: Važnost sjena kod određivanja položaja objekta u prostoru

Zbog važnosti sjena za realistično prikazivanje i izazova koji predstavljaju, ovo područje je dobilo puno pozornosti, posebno u kontekstu stvarnog vremena, gdje u isto vrijeme postoji težnja prema realizmu, ali i ograničenja u vidu željene brzine algoritma. Iako danas postoje grafičke kartice koje omogućuju djelovanje algoritma praćenja zrake u stvarnom vremenu, koji između ostalog daje i fotorealistične sjene, u velikoj većini slučajeva on je i dalje računski preskup. Trenutno najbolji omjer brzine i kvalitete sjena, a time i najkorišteniji način prikaza sjena u industriji, daje algoritam mape sjene, točnije njegove razne poboljšane varijacije što je ujedno i tema ovog rada.

2. Mapa sjene

Algoritam mape sjene (engl. *Shadow Mapping*) je prvi put opisan u [12]. U nastavku je objašnjen princip koji stoji iza algoritma.

2.1. Princip mape sjene

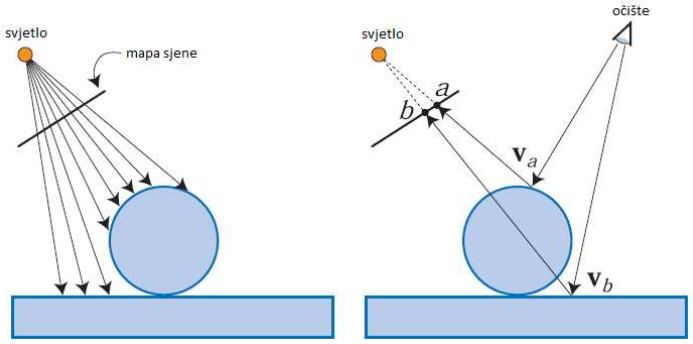


Slika 2.1: Tok transformacija za iscrtavanje dubinske mape iz pogleda svjetla. Isti takav tok se koristi i kod uobičajenog iscrtavanja, uz matrice pogleda i projekcije kamere umjesto svjetla.

Glavno opažanje na kojem se temelji algoritam je da izvor svjetlosti "vidi" sve osvijetljene točke u sceni, odnosno sve skrivene točke su u sjeni za taj izvor svjetlosti. Slika 2.2 pojednostavljeno prikazuje to opažanje. Kako bi pronašli sve vidljive točke, prvi korak je generiranje dubinske mape (engl. depth map) iz pogleda izvora svjetlosti. Takva dubinska mapa se naziva mapa sjene (engl. shadow map). U ovom koraku se ne računa osvjetljenje. Svaki slikovni element dubinske mape sadrži udaljenost najbliže vidljive točke od izvora svjetlosti. Ta udaljenost se u mapu sprema kao normalizirana z-koordinata (engl. normalized device coordinates)[1]. Na slici 2.1 se vidi da je za normalizirane koordinate potrebno pronaći odgovarajuće matrice pogleda i projekcije svjetla. Koja se vrsta projekcije koristi i na koji se način računa matrica pogleda ovisi o vrsti izvora svjetlosti.
Grafički hardver omogućuje vrlo brzo generiranje takvih mapa s obzirom da se isti takav mehanizam koristi kod određivanja vidljivosti.

[1] U DirectX API-ju ta je vrijednost iz intervala $[0, 1]$ dok je u OpenGL-u iz $[-1, 1]$. x i y normalizirane koordinate su iz intervala $[-1, 1]$ kod oba API-ja

Drugi korak algoritma je iscrtavanje scene iz očišta, tj. kamere. Za svaki rasterizirani fragment transformiramo njegovu globalnu poziciju (engl. world space position) u koordinate prostora projekcije svjetla (engl. light clip space) koristeći iste one matrice izračunate za prvi korak. Perspektivnim dijeljenjem se dobiju normalizirane koordinate za taj fragment. Sada je dovoljno usporediti dobivenu z-koordinatu s vrijednošću spremljenom u mapi sjene - ako je dobivena vrijednost veća od one u mapi, ta točka se nalazi u sjeni, u suprotnom je osvjetljena (Slika 2.2). Dobivene x i y normalizirane koordinate određuju poziciju s kojom uzorkujemo mapu sjene. Te koordinate su obje iz intervala $[-1, 1]$, a koordinate teksture iz $[0, 1]$ što znači da je potrebno obaviti preslikavanje $x \mapsto \frac{x+1}{2}$ nad normaliziranim koordinatama prije uzorkovanja teksture [2]. Slika 2.3 prikazuje jednostavnu scenu s ravninom i sferom te njenu pripadnu mapu sjene.

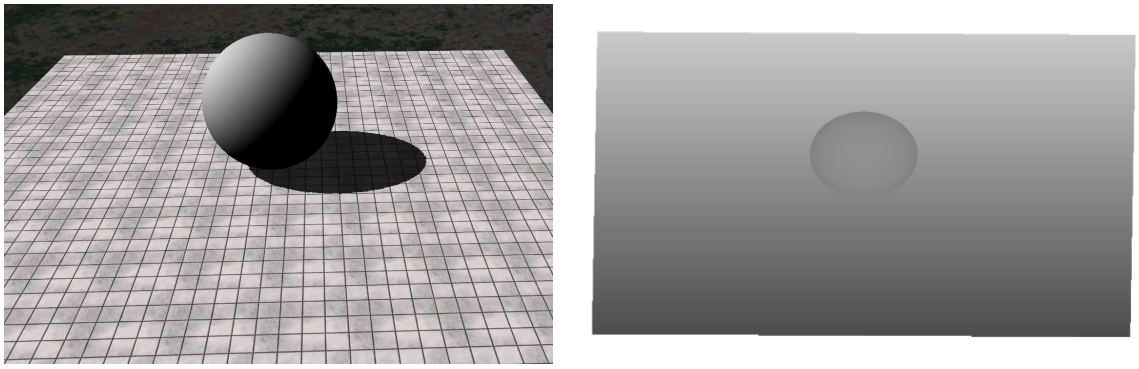


Slika 2.2: Princip iza algoritma mape sjene. Lijeva strana slike (pojednostavljeno) prikazuje generiranje mape sjene, dok desna strana slike prikazuje testiranje jesu li točke $\boldsymbol{v}_a$ i $\boldsymbol{v}_b$ u sjeni. Udaljenost točke $\boldsymbol{v}_a$ od svjetla iznosi približno isto kao odgovarajuća vrijednost u mapi sjene pa je ta točka osvjetljena. Udaljenost točke $\boldsymbol{v}_b$ od svjetla je veća od odgovarajuće vrijednosti u mapi sjene pa se za tu točku zaključi da je u sjeni. (slika iz [5])

2.2. Različite vrste svjetla

Konstrukcija matrica potrebnih za generiranje dubinske mape se razlikuje ovisno o vrsti izvora svjetla za koji se provodi postupak.

[2] U DirectX API-ju potrebno je dodatno obaviti preslikavanje $y \mapsto 1 - y$ nad y koordinatom zato što je ishodište teksture u gornjem lijevom kutu



Slika 2.3: Primjer jednostavne scene i generirane dubinske mape

Usmjereno svjetlo

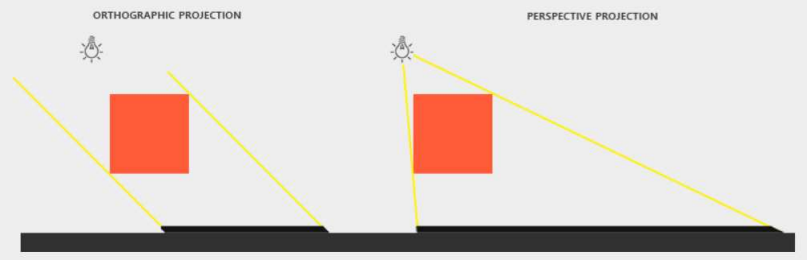
Usmjereno svjetlo po definiciji nema poziciju, ali je ona potrebna kako bi se izračunala matrica pogleda. Jedan od načina je postaviti poziciju svjetla dovoljno daleko da volumen pogleda koji je definiran matricom projekcije obuhvati sve objekte u sceni. Matrica projekcije je ortografska i u ovom slučaju se relativno lako izračuna na temelju omeđujućeg kvadra ili sfere scene. Izračun parametara za ortografsku projekciju mora biti izračunat u prostoru svjetla što je jednostavnije u slučaju sfere jer je invarijantna na rotacije za razliku od kvadra. Ovakav način prikladan je za male i jednostavne scene.
Drugi način je koristiti krnju piramidu ili volumen pogleda kamere (engl. camera view frustum). Prvo se izračuna centroid krnje piramide te se pozicija svjetla izračuna koristeći formulu:

$$\boldsymbol{p} = \boldsymbol{c} - \lambda \boldsymbol{d}, \quad \lambda > 0, \lambda \in \mathbb{R}$$

$\boldsymbol{c}$ je izračunati centroid, a $\boldsymbol{d}$ je smjer usmjerenog svjetla. Ako je parametar λ prenizak, postoji mogućnost da volumen pogleda ne obuhvati sve objekte koji bacaju sjenu u vidljivi dio scene, a s previsokim λ dolazi do lošije rezolucije mape sjene. Ovim načinom dolazi i do pojave zvane "plivanje sjena" (engl. shadow swimming, shadow shimmering) koja se javlja zbog translacije i rotacije kamere. Taj problem se može riješiti uz cijenu gubitka preciznosti mape sjene. Matrica projekcije se računa na temelju omeđujućeg kvadra ili sfere krnje piramide pogleda kamere. Kaskadne mape sjena koriste isključivo ovaj način u konstrukciji svojih mapa.

Točkasto usmjereno svjetlo

Točkasto usmjereno svjetlo ili reflektor (engl. spotlight) je najjednostavniji slučaj. Postupak je vrlo sličan kao u slučaju usmjerenog svjetla. Koristi se perspektivna projekcija (vidi sliku 2.4) umjesto ortografske, a pozicija i smjer svjetla su zadane definicijom što znači da nije potrebno provoditi dodatne postupke kako bi se izračunala matrica pogleda.



Slika 2.4: Usmjereno svjetlo koristi ortografsku projekciju, a reflekor koristi perspektivnu projekciju [11]

Točkasto svjetlo

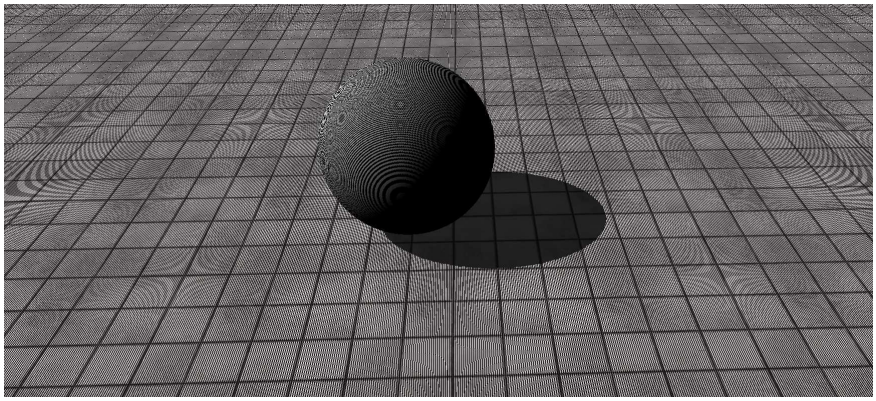
Točkasti izvori za razliku od reflektora emitiraju svjetlost u svim smjerovima pa nije moguće koristiti sličan pristup jer ne postoji nijedna perspektivna projekcija koja pokriva kut od $360°$. Uobičajeni pristup točkastim izvorima je računanje šest različitih volumena pogleda, svaki za jednu stranu kocke. Svih šest volumena pokriva prostorni kut od $\frac{2\pi}{3}$sr. Tijekom proračuna sjene, točka se projicira na jednu od šest mapi sjena. Najčešće se koristi kocka tekstura (engl. cube map), struktura standardna na današnjim GPU-ovima. Najjednostavniji, ali i najneefikasniji način je sve mape sjene generirati odvojeno, tj. ponavljamo postupak šest puta, ali svaki put s drugim matricama pogleda. Moguće optimizacije su korištenje geometrijskog sjenčara kako bi se smanjio broj poziva ili parabolično mapiranje koje koristi samo dvije teksture ([5]).

2.3. Nedostatci i poboljšanja

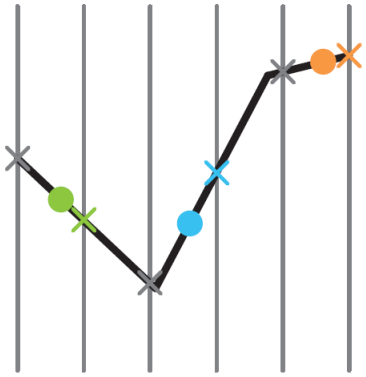
Akne sjena

Akne sjena (engl. *Shadow acne*) su čest problem koji je vidljiv u naivnoj implementaciji mape sjene, a predstavljaju pojavu u kojoj objekt neispravno baca sjenu sam na sebe (engl. *self-shadow aliasing*). Slika 2.5 prikazuje primjer akne sjeni. Dva

su uzroka ove pojave. Jedan je ograničena numerička preciznost brojeva s pomičnim zarezom, a drugi je diskretna priroda mape sjene jer skoro nikad neće postojati jednoznačno preslikavanje između fragmenata i elemenata mape sjene što znači da ćemo pri usporedbi zapravo koristiti kvantizirane vrijednosti, a ne stvarne (Slika 2.6).



Slika 2.5: Pojava akni sjena u naivnoj implementaciji algoritma



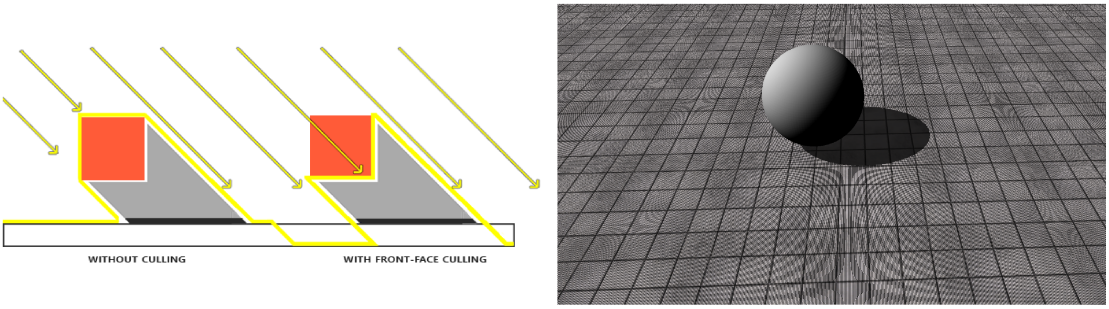
Slika 2.6: Tamna linija prikazuje površinu objekta, vertikalne sive linije predstavljaju elemente mape sjene u kojima su spremljene dubine. Tri obojana kružića predstavljaju točke za koje određujemo jesu li u sjeni ili ne. Vrijednosti koje dobijemo uzorkovanjem mape sjene su obojani križići. Plava i narančasta točka će greškom biti u sjeni zato što su udaljeniji od svjetla nego referentne vrijednosti u mapi sjene ([1])

.

Jedan od mogućih pristupa je odbaciti prednje poligone i u dubinsku mapu pisati dubine samo stražnjih poligona kao na slici 2.7 [3].

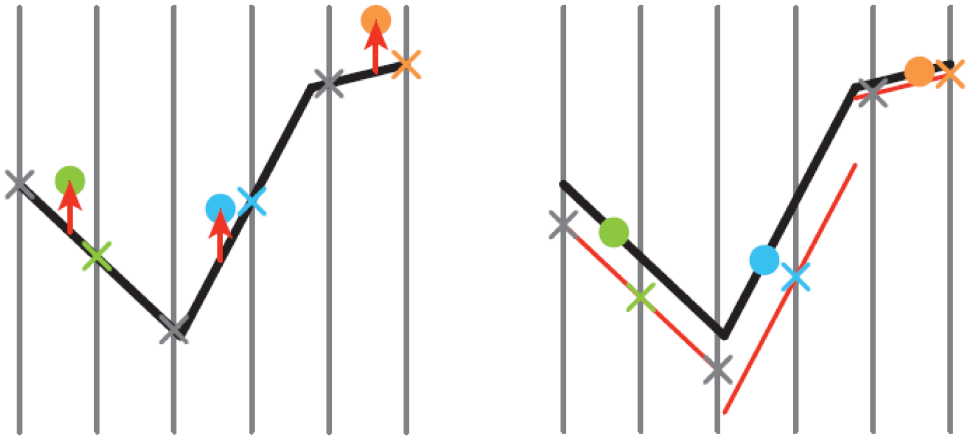
Ovaj pristup ne funkcionira kada se stražnji poligon poklapa s prednjim ili ako je udaljenost između njih zanemariva. To je i slučaj na desnoj strane slike 2.7 gdje su na

[3] Kod uobičajenog iscrtavanja je obrnuto, odbacuju se stražnji poligoni jer ih ne vidimo, a prednji se iscrtavaju



Slika 2.7: Utjecaj odbacivanja prednjih poligona na akne sjene

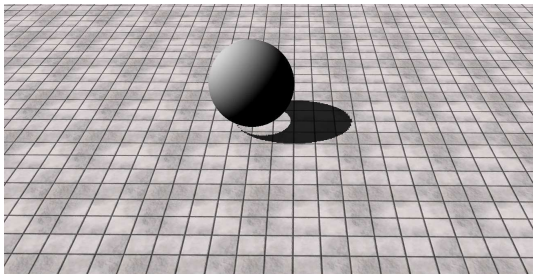
sferi uklonjene akne sjene, ali su na ravnini i dalje prisutne.
Ipak najčešći i sigurniji pristup je korištenje male vrijednosti (engl. bias) koja se doda vrijednostima u mapi sjene te time izbjegnu nepreciznosti kod uspoređivanja dvaju brojeva s pomičnim zarezom. Na slici 2.8 se vidi ideja iza dodavanja biasa. Moguće je i obrnuto, umjesto zbrajanja s vrijednostima u mapi, oduzimati od trenutne vrijednosti. Korištenje prevelikog biasa vodi do "curenja svjetla" (engl. *light leaking*), obično na



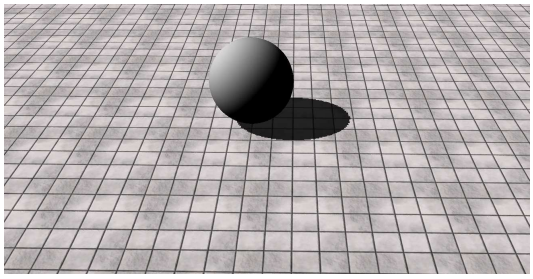
Slika 2.8: Slika lijevo prikazuje dodavanje konstantnog biasa za cijeli objekt. Plava točka će i dalje greškom biti u sjeni. Rješenje problema je koristiti bias koji ovisi o nagibu, tj. kutu između smjera svjetla i normale u točki koju sjenčamo. Taj pristup je prikazan na slici desno. ([1]).

mjestu gdje se objekt koji baca sjenu i površina koja prima sjenu dodiruju. U tom slučaju sjene izgledaju odvojeno od objekta pa se ova pojava još naziva i *Peter Panning* (vidi sliku 2.9).

Određivanje vrijednosti biasa nije lagan zadatak i općenito ta vrijednost ne bi trebala biti konstantna za cijelu scenu (slika 2.8), već bi trebala biti veća što je kut koji zatvara normala površine i smjer svjetla veći. Iako je moguće direktno koristiti neku



(a) Prevelik bias uzrokuje odvajanje sjene od objekta



(b) Ispravan bias uklanja akne sjene i ne uzrokuje odvajanje sjene od objekta

Slika 2.9: Usporedba prevelikog i ispravnog biasa

funkciju bias $\sim \angle(\boldsymbol{n}, \boldsymbol{d})$ poput $\epsilon \cdot \tan(\angle(\boldsymbol{n}, \boldsymbol{d}))$ i manualno dodavati bias kod usporedbe bolje je koristiti API funkcije i varijable predviđene za to. DirectX s varijablama DepthBias, SlopeScaledDepthBias i DepthBiasClamp iz strukture D3D11_RASTERIZER_DESC i OpenGL/Vulkan s funkcijama glPolygonOffset/vkCmdSetDepthBias podržavaju takav način računanja. Na primjer, u DirectX API-ju se za $32-$bitni realni spremnik dubine konačni bias izračuna na idući način:

$$\text{Bias} = \text{DepthBias} \cdot 2^{\text{exponent}(\max z)-23} + \text{SlopeScaledDepthBias} \cdot \max\left(\left|\frac{\partial z}{\partial x}, \frac{\partial z}{\partial y}\right|\right)$$

te se još provjeri je li dobivena vrijednost po apsolutnoj vrijednosti veća od maksimalne dozvoljene - DepthBiasClamp. Isti ovakav proračun koristi i Vulkan, samo s drugim imenima varijabli. Bias se zatim doda svakoj z-vrijednosti u normaliziranim koordinatama što znači da se mapa sjena generira s ovim mehanizmom, ostalo iscrtavanje mora to isključiti.

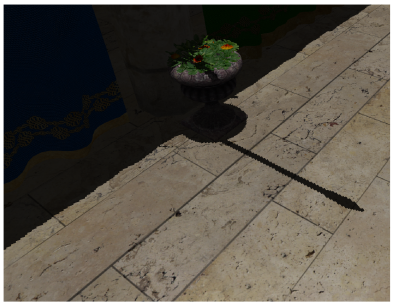
Alias-efekt

Algoritam mape sjene je zapravo diskretizacijski postupak što ga čini sklonim neželjenom alias-efektu koji se u ovom postupku manifestira kao nazubljeni rubovi sjena objekata. Više je uzroka alias-efekta u slučaju mape sjene. Kod postupka generiranja mape sjene, vrlo često se više rasteriziranih fragmenata preslikava u jedan te isti element mape sjene što znači da su vrlo često svi ti fragmenti ili u sjeni ili nisu, a to dovodi do gubitka visokofrekventnih detalja. Takav nesrazmjer fragmenata i elemenata mape sjene se naziva poduzorkovanje jer frekvencija uzorkovanja nije dovoljno visoka. Alias-efekt uzrokovan poduzorkovanjem se još naziva prealias. Razlikuju se dva tipa poduzorkovanja: perspektivno i projekcijsko poduzorkovanje. Jedan jednostavan, ali relativno skup način kojim se ublažava alias-efekt je povećanje rezolucije

(a) Veličina mape 512×512

(b) Veličina mape 1024×1024

(c) Veličina mape 2048×2048



(d) Veličina mape 4096×4096

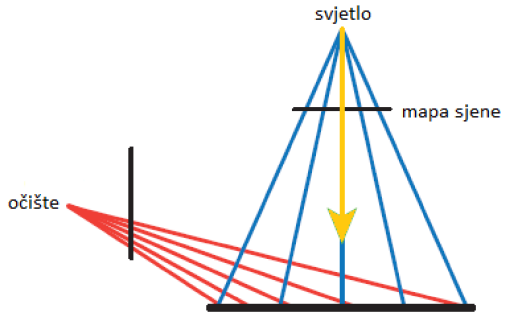
Slika 2.10: Utjecaj veličine mape sjene na alias-efekt. Alias-efekt se ne može ukloniti, već samo ublažiti.

mape sjene jer se time smanjuje šansa da dva fragmenta dijele isti element mape sjene. Utjecaj veličine mape sjene može se vidjeti na slici 2.10. U postupku iscrtavanja scene, točnije kod ponovnog uzorkovanja mape sjene (engl. *resampling*), dolazi do drugog tipa alias-efekta koji se još naziva i postalias i on je uzrokovan greškom rekonstrukcije signala.

Perspektivno poduzorkovanje

Perspektivno poduzorkovanje je izraženiji je kod fragmenata bliže očištu što se može vidjeti na slici 2.11. Sad je jasno otkud takvo ime, perspektivno poduzorkovanje uzrokuje perspektivna priroda kamere.

Zbog velikog utjecaja na kvalitetu sjena razvijeni su razni algoritmi koje nastoje ublažiti utjecaj perspektivnog poduzorkovanja. Najpopularniji algoritam u toj borbi su kaskadne mape sjena (CSM, PSSM) koji spada u particijske algoritme te će biti obrađen kasnije.



Slika 2.11: Fragmenti bliži očištu imaju veću pogrešku perspektivnog poduzorkovanja

Projekcijsko poduzorkovanje

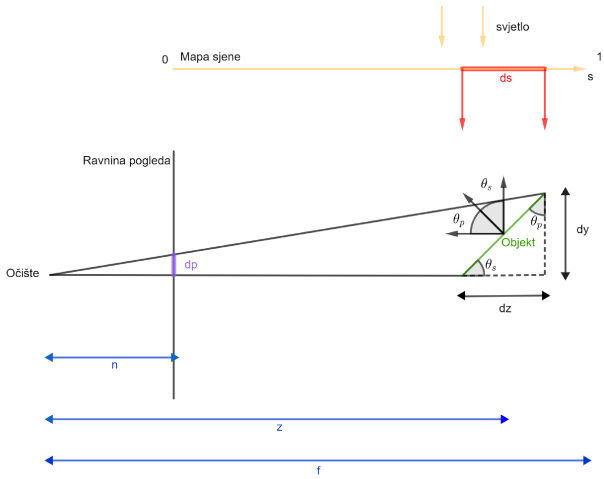
Drugi tip poduzorkovanja koji se javlja kod mape sjene je projekcijsko poduzorkovanje. Najveća greška se javlja za površine skoro paralelne sa smjerom svjetla iz čega slijedi da on ovisi o orijentaciji geometrije u odnosu na smjer svjetla. Kako bi se smanjio utjecaj, potrebno je lokalno povećati gustoću uzorkovanja u područjima gdje je to potrebno što zahtjeva detaljnu analizu geometrije scene (obično se koriste strukture poput četvero stabala). To je posebno zahtjevno za velike i složene scene. Iako postoje algoritmi koji reduciraju utjecaj projekcijskog poduzorkovanja oni zbog svoje računske složenosti nisu prikladni za rad u stvarnom vremenu te se ne razmatraju u ovom radu.

Greška rekonstrukcije

Vrlo često kod ponovnog uzorkovanja mape sjene, točka uzorkovanja se neće poklopiti sa središtem elementa mape. U tom slučaju je potrebno provesti rekonstrukciju na temelju dostupnih vrijednosti koristeći rekonstrukcijski filter. Najjednostavniji, ali najgori rekonstrukcijski filter je interpolacija najbližeg susjeda. Upravo ova greška uzrokuje nazubljene rubove sjena. Za smanjivanje greške se obično provodi bilinearna interpolacija što nije trivijalno u slučaju algoritmu mape sjene. Međutim, čak ni idealni filter ne može nadomjestiti detalje izgubljene zbog poduzorkovanja. Algoritmi koji koriste filtriranje s ciljem smanjivanja greške rekonstrukcije su obrađeni u idućem poglavlju. Najpoznatiji predstavnik ove skupine algoritama je PCF.

Analiza greške poduzorkovanja

U ovom odjeljku se provodi znatno pojednostavljena analiza pogreške poduzorkovanja kako bi se donekle opravdale tvrdnje u prošlim odjeljcima - perspektivno poduzorkovanje je najveće za točke blizu očišta, a projekcijsko za geometriju paralelnu sa smjerom svjetlosti. Vidi sliku 2.12 za pojednostavljenu skicu na kojoj se temelji analiza. Gre-



Slika 2.12: Analiza greške poduzorkovanja u mapi sjene

šku poduzorkovanja uzrokuje preslikavanje više slikovnih elemenata na jedan te isti element mape sjene. Analiza se provodi tako da se kroz jedan element mape sjene veličine ds pošalju zrake svjetla te se površina objekta koju pogađaju te zrake preslika na ravninu pogleda na površinu dp. Ako je dp veće od veličine jednog slikovnog elementa tada dolazi do poduzorkovanja. Pretpostavka je da postoji lokalna parametrizacija mape sjene, tj. $s \in [0, 1]$ za $z \in [n, f]$ što dodatno znači da mapa sjene sadrži isključivo vidljivi dio scene što obično nije istina. Lokalna parametrizacija omogućuje usporedbu drugačijih tipova parametrizacija. Tada je npr. za uniformnu parametrizaciju $dz = (f - n)\, ds$.

Na slici 2.12, zrake svjetlosti pogađaju mali rub koji je duljine:

$$\frac{dz}{\cos\theta_s}$$

Iz te vrijednosti se opet iz pravokutnog trokuta izračuna dy:

$$dy = \frac{dz\cos\theta_p}{\cos\theta_s}$$

Iz sličnih trokuta:

$$\frac{dy}{z} = \frac{dp}{n} \Rightarrow dp = \frac{n}{z}\,dy = \frac{n}{z}\frac{dz\cos\theta_p}{\cos\theta_s}$$

Budući da je $z = f(s)$ (točni izraz ovisi kakva se parametrizacija koristi) izraz za grešku poduzorkovanja se definira kao omjer $\frac{dp}{ds}$ kako bi se lakše usporedile razne parametrizacije:

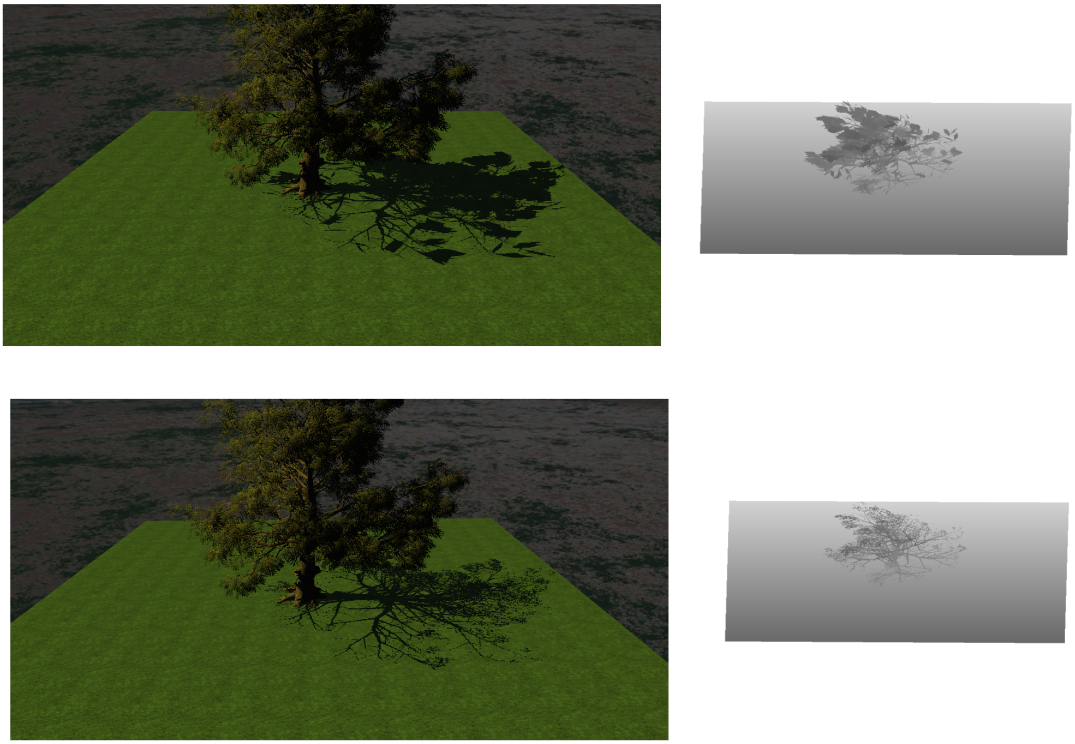
$$\frac{dp}{ds} = \frac{n}{z}\frac{dz}{ds}\frac{\cos\theta_p}{\cos\theta_s}$$

U ovoj formuli $\frac{n}{z}\frac{dz}{ds}$ predstavlja perspektivno poduzorkovanje. Omjer $\frac{dz}{ds}$ je konstantan za uniformne mape sjene i iznosi $(f - n)$ pa nije toliko značajan. Izraz $\frac{n}{z}$ pokazuje da objekti bliže očištu imaju izraženiji perspektivni alias-efekt budući da oni manju vrijednost z. Vrijedi napomenuti da postoji skupina algoritama koji promjenom parametrizacije smanjuju omjer $\frac{dz}{ds}$ pa time i perspektivno poduzorkovanje. Ovakva analiza ne uzima u obzir veličinu mape sjene već samo lokalnu parametrizaciju. U detaljnijoj analizi poput u [8] bi se izraz za veličinu mape sjene pojavio u nazivniku odakle se dođe do zaključka da povećanje mape sjene također smanjuje grešku poduzorkovanja. $\frac{\cos\theta_p}{\cos\theta_s}$ predstavlja projekcijsko poduzorkovanje te se slaže s komentarom da je projekcijska greška poduzorkovanja najveća za površine skoro paralelne sa smjerom svjetla jer je tada $\theta_s \approx \frac{\pi}{2} \Rightarrow \cos\theta_s \approx 0 \Rightarrow \frac{\cos\theta_p}{\cos\theta_s} \to \infty$.

2.4. Transparentni objekti

U ovoj osnovnoj verziji algoritam ne podržava sjene transparentnih objekata, poput lišća ili trave koji često koriste geometriju pravokutnika s difuznom teksturom čiji je alfa kanal jednak nula (ili vrlo niskom broju) gdje god taj pravokutnik treba biti proziran. Primjer jedne takve teksture je dan na slici 2.13.

Slika 2.13: Primjer transparentne teksture



Slika 2.14: Scena s ravninom te stablom sa transparentnim lišćem.
Gornji red: prikaz scene i dubinske mape bez podrške za transparentne objekte
Donji red: prikaz scene i dubinske mape s podrškom za transparentne objekte

Na sreću, dodavanje podrške za takve objekte je lagano (barem za one koji informaciju transparentnosti čuvaju u alfa kanalu teksture). Za vrijeme generiranja mape sjene, prije pisanja dubine, alfa kanal teksture se provjeri u sjenčaru fragmenta/piksela te se na temelju vrijednosti odbaci ili prihvati. Pisanje dubine se obavlja automatski. Slika 2.14 prikazuje razliku između mapi sjena s i bez podrške za transparentne objekte.

3. Filtrirane mape sjena

U ovom poglavlju se razmatraju metode filtriranja mape sjene. Njihova glavna svrha je smanjenje greške rekonstrukcije što čini artefakte uzrokovane poduzorkovanjem manje izraženim, a to se postigne zaglađivanjem rubova sjena. Istovremeno se dobiju sjene koje podsjećaju na fizikalno temeljene meke sjene uz puno manju računsku složenost. Jednostavna implementacija te relativno dobre performanse i rezultati čine ovu skupinu metoda od velike važnosti u industriji.
U kontekstu filtriranja mapa sjena, filtriranje se smatra bilo koji pristup u kojem se na temelju više uzoraka mape sjene računa je li neka točka u sjeni (te koliko je u sjeni!). Međutim, zbog binarne prirode testa u mapi sjene (točka je u sjeni ili nije) direktno filtriranje dubinske mape ne daje željene rezultate.

3.1. Direktno filtriranje mape sjene

Označimo funkciju dubine sa $z(\mathbf{t})$ gdje $\mathbf{t} = (s,t)$ označava koordinate u prostoru teksture. Osnovni algoritam koristi funkciju usporedbe $f(\mathbf{t},\tilde{z}) = s(z(\mathbf{t}),\tilde{z})$ kako bi odredio je li fragment s dubinom u prostoru svjetlosti $\tilde{z}$ u sjeni ili ne. Uobičajena funkcija usporedbe je Heavisideova step funkcija $s(z,\tilde{z}) = \mathcal{H}(z-\tilde{z})$ koja je definirana kao

$$\mathcal{H}(z) = \begin{cases} 0 & \text{ako je } z < 0 \\ 1 & \text{inače} \end{cases}$$

Zamućivanjem vrijednosti spremljene u mapi sjene te korištenje funkcije usporedbe nad zamućenim vrijednostima ne daje željene rezultate zato što je izlaz funkcije

i dalje binaran - točka je u sjeni ili nije [1].

$$f_{\text{filter}}(\mathbf{t}, \tilde{z}) = \mathcal{H}\left(\sum_{t_i \in \mathcal{K}} k(\mathbf{t}_i - \mathbf{t}) z(\mathbf{t}_i) - \tilde{z}\right) \in \{0, 1\}$$

3.2. Algoritam PCF

Bolji pristup je filtrirati izlaze funkcije usporedbe, a ne dubine spremljene u mapi. Ovaj pristup se naziva engl. *percentage-closer filtering* ili skraćeno PCF. Matematički zapis funkcije usporedbe za PCF je:

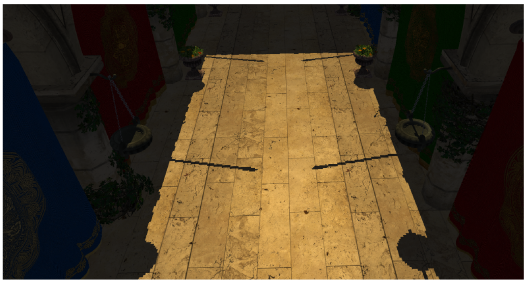
$$f_{\text{pcf}}(\mathbf{t}, \tilde{z}) = \sum_{t_i \in \mathcal{K}} k(\mathbf{t}_i - \mathbf{t}) \mathcal{H}\left(z(\mathbf{t}_i) - \tilde{z}\right)$$

gdje je $\mathcal{K}$ jezgra filtra, a k funkcija jezgre koja određuje težine uzoraka. Zbog vrlo male razlike u implementaciji u odnosu na osnovni algoritam mape sjene ova metoda filtiranja je izrazito popularna. Međutim, značajan nedostatak je činjenica da se funkcija usporedbe treba izračunati prije samog filtriranja što onemogućuje filtriranje prije samog iscrtavanja scene. Iz istog razloga nije moguće generirati mip-mape za mapu sjene. Npr. za $n \times n$ jezgru filtera, potrebno je uzorkovati teksturu na GPU n^2 puta za svaki rasterizirani fragment, a uzorkovanje teksture je jedna od najskupljih operacija. Također, PCF ne rješava česte probleme mape sjene poput akni sjena.

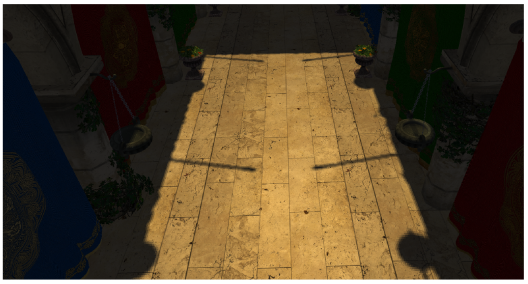
Implementacije PCF-a najčešće razlikujemo po korištenoj jezgri filtra te funkciji jezgre. Najčešće jezgre filtra su pravilna rešetka $n \times n$, Gaussova jezgra te Poissonov disk, dok je najčešća funkcija jezgre konstantna funkcija, tj. $k(\mathbf{t}) = \frac{1}{|\mathcal{K}|}, \forall \mathbf{t} \in \mathcal{K}$. S konstantnom funkcijom jezgre i pravilnom rešetkom $n \times n$ postoji $n^2 + 1$ različitih izlaza funkcija usporedbe. Korištenjem istih lokacija uzorkovanja za svaki fragment može dovesti do uočljivih uzoraka. Takvi artefakti se mogu izbjeći rotacijom jezgre filtra za svaki fragment, a kut rotacije može biti spremljen u teksturi, izračunat pomoću jeftine pseudoslučajne funkcije ili jednostavno zapisan u kodu sjenčara kao polje vrijednosti. I OpenGL/Vulkan i DirectX 10+ podržavaju hardversku implementaciju 2×2 PCF-a u jednoj operaciji koristeći bilinearni filter prilagođen za ovu situaciju. To se može dodatno iskoristiti za lakše ostvarivanje filtera višeg reda. Slika 3.1 prikazuje usporedbu sjena dobivenih osnovnim i PCF algoritmom.

Budući da je najveći nedostatak PCF-a nemogućnost obavljanja filtriranja direktno na mapu sjene, već samo u prostoru slike (engl. *screen space*, *image space*), razvijene su

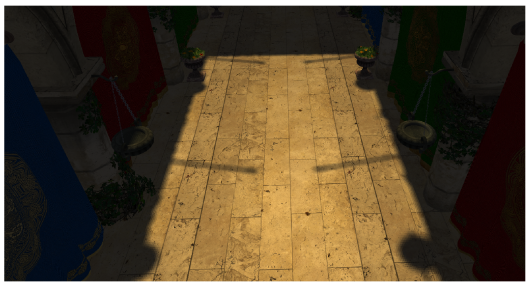
[1] Jedna od ideja je koristiti neku glatku verziju Heavisideove funkcije, ali ispostavlja se da takav način nije prikladan za fotorealistično iscrtavanje



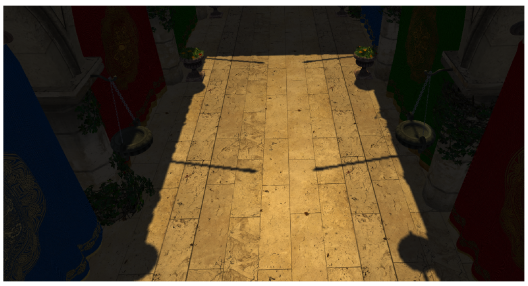
(a) Bez filtriranja



(b) PCF 3×3 pravilna rešetka



(c) PCF 5×5 pravilna rešetka



(d) PCF Poissonov disk sa 64 uzorka

Slika 3.1: Usporedba PCF algoritma s osnovnim algoritmom. Veličina mape sjene je u sva četiri slučaja 2048×2048

metode koje zaobilaze to ograničenje te su neke od njih predstavljene u idućim odjeljcima. Tada će biti moguće i generiranje mip-mapa(što je također vrsta filtriranja!) za mapu sjene što dosad nije bilo moguće. U nastavku se takvo "ranije" filtriranje naziva predfiltriranje.

3.3. Algoritam VSM

Mape sjene temeljene na varijanci ili skraćeno VSM (engl. *Variance Shadow Maps*) koriste jednostavnu statističku formulu kako bi omogućili predfiltriranje mape sjene ([4]). Umjesto korištenja funkcije usporedbe kao PCF, procjenjuje se distribucija dubina unutar jezgre filtra. Za procjenu je potrebno izračunati srednju vrijednost μ i varijancu σ^2 uzoraka unutar jezgre filtra. Prva razlika u odnosu na PCF (pa time i na osnovni algoritam) je zapisivanje i kvadrata dubine, obično u drugi kanal elementa mape. Kako bi se mogle izračunati tražene vrijednosti kada za to dođe vrijeme dovoljno je primijeniti običan filter zamućivanja direktno na mapu sjene. Velika prednost VSM-a proizlazi iz činjenice da je filter zamućivanja separabilan čime se značajno smanji broj uzorkovanja mape. Za $n \times n$ rešetku, složenost jednoprolaznog filtera je kvadratna $\mathcal{O}(n^2)$, dok je za separabilan filter ta složenost linearna $\mathcal{O}(2n) = \mathcal{O}(n)$.

Nakon filtriranja idući korak je pronaći μ i σ^2. Taj proračun se obično odvija u prostoru slike, netom prije proračuna osvjetljenja. Filtrirana mapa sada sadrži prvi moment M_1 u prvom kanalu te drugi moment M_2 u drugom kanalu. Izračun traženih vrijednosti je sada trivijalan:

$$\begin{aligned} M_1 &= \sum_{i \in \mathcal{K}} x_i p_i \\ M_2 &= \sum_{i \in \mathcal{K}} x_i^2 p_i \\ \mu &= M_1 \\ \sigma &= M_2 - M_1^2 \end{aligned}$$

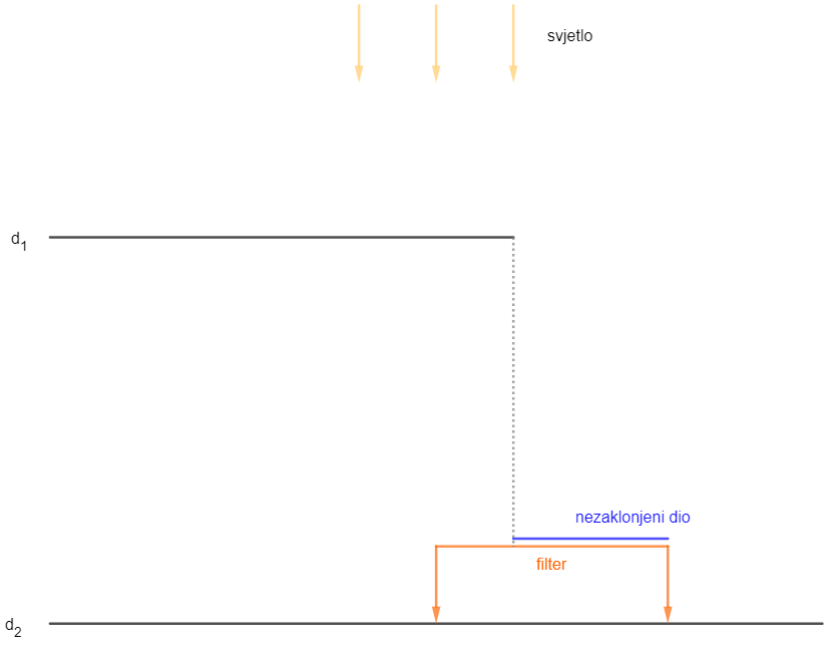
Kako bi se sad na temelju nepoznate distribucije, ali poznavajući srednju vrijednost i varijancu te distribucije, procijenila vidljivost neke točke koristi se Čebiševljeva nejednakost:

Čebiševljeva nejednakost. Neka je z slučajna varijabla s očekivanjem μ i varijancom σ. Tada za $\tilde{z} > \mu$ vrijedi

$$P(z \geq \tilde{z}) \leq p_{\text{max}}(\tilde{z}) = \frac{\sigma^2}{\sigma^2 + (\tilde{z} - \mu)^2}$$

U našem slučaju $\tilde{z}$ je dubina u prostoru svjetlosti promatranog fragmenta, a μ su srednja vrijednost i varijanca jezgre filtriranja za taj fragment spremljeni u mapi sjene. U prijevodu, najviše $p_{\text{max}}(\tilde{z})$ točaka u jezgri filtera nije bliže izvoru svjetlosti od promatranog fragmenta. To znači da ispravna sjena nikad ne može biti svjetlija već samo tamnija od ove procjene. U slučaju da dubina fragmenta nije veća od srednje vrijednosti μ, Čebiševljeva nejednakost ne vrijedi te se tada uzima $P(z \geq \tilde{z}) = 1$. Kada $\tilde{z} \approx \mu$, može doći do numeričkih problema ako je $\sigma^2 \approx 0$ pa se obično koristi neka minimalna vrijednost za varijancu $\sigma^2_{\min}$.

Lauritzen i Donnelly su u [4] pokazali da za poseban slučaj paralelnih ravnina u kojem jedna baca sjenu, a druga prima, VSM daje jednake rezultate kao PCF. Neka je dubina prve ravnine d_1, a druge d_2. Neka je p udio filtra koji nije zaklonjen (Vidi sliku 3.2). Tada je

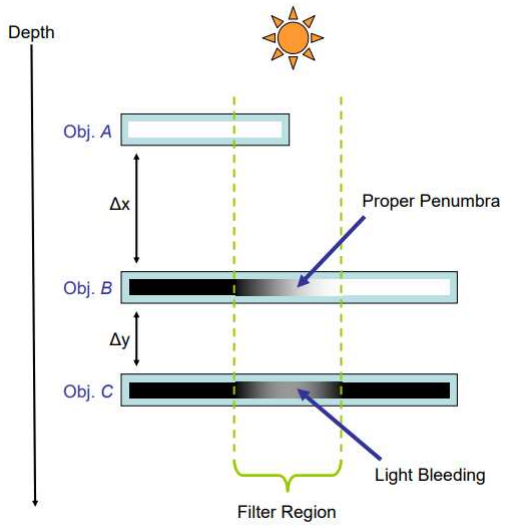


Slika 3.2: Paralelne ravnine na dubinama d_1 i d_2

$$
\begin{aligned}
\mu = M_1 &= pd_2 + (1-p)d_1 \\
M_2 &= pd_2^2 + (1-p)d_1^2 \\
\sigma^2 &= M_2 - M_1^2 = (p-p^2)(d_2-d_1)^2 \\
p_{\max}(d_2) &= \frac{\sigma^2}{\sigma^2 + (\mu - d_2)^2} \\
&= \frac{(p-p^2)(d_2-d_1)^2}{(p-p^2)(d_2-d_1)^2 + (1-p)^2(d_2-d_1)^2} \\
&= \frac{p-p^2}{1-p} = p
\end{aligned}
$$

Curenje svjetla

Jednostavnost i brzina VSM-a ima svoju cijenu jer u nekim slučajevima dolazi od izraženog curenja svjetla (engl. *light leaking*). Iako Čebiševljeva nejednakost daje gornju među, ta gornja međa nije uvijek dobra aproksimacija. Slika 3.3 prikazuje situaciju u kojoj dolazi do curenja svjetla. Neka se objekti nalaze redom na dubinama a, b, c. Objekt C nije vidljiv iz pogleda svjetlosti. Vrijednosti zapisane u mapi sjene za



Slika 3.3: Curenje svjetla u algoritmu VSM ([4])

točku u središtu regije filtera na površini C

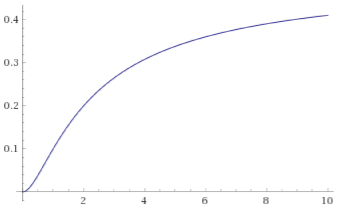
$$M_1 = \frac{a+b}{2}$$
$$M_2 = \frac{a^2+b^2}{2}$$

Tada se lako izračuna

$$\mu = \frac{a+b}{2}$$
$$\sigma^2 = \frac{(b-a)^2}{4}$$

Čebiševljevom nejednakošću se dobije

$$\begin{aligned} p_{\max}(c) &= \frac{\sigma^2}{\sigma^2 + (\mu - c)^2} \\ &= \frac{(b-a)^2}{(b-a)^2 + (a+b-2c)^2} \\ &= \frac{(b-a)^2}{(b-a)^2 + (-2c+2b+a-b)^2}, \quad b-a = \Delta x, c-b = \Delta y \\ &= \frac{\Delta x^2}{\Delta x^2 + (2\Delta y + \Delta x)^2} \\ &= \frac{\Delta x^2}{2\Delta x^2 + 4\Delta x \Delta y + 4\Delta y^2} \\ &= \frac{t^2}{2t^2 + 4t + 4}, \quad t = \frac{\Delta x}{\Delta y} > 0 \end{aligned}$$



Slika 3.4: Funkcija $f(t) = \frac{t^2}{2t^2+4t+4}$

Ispravna vrijednost vidljivosti je nula, međutim zato što je $t > 0$ tada je i $p_{\max}(c) > 0$. Sa slike 3.4 se vidi da funkcija $f(t) = \frac{t^2}{2t^2+4t+4}$ na intervalu $[0, \infty)$ monotono raste te da teži $\lim_{t\to\infty} f(t) = 0.5$. Veći omjer $t = \frac{\Delta x}{\Delta y}$, odnosno veći Δx i/ili manji Δy, uzrokuje veće curenje svjetlosti. Proračun je izveden za jednostavan slučaj, ali zaključak stoji i u općenitom slučaju.

Slika 3.5: Curenje svjetla kod VSM-a na slici lijevo te redukcija curenja svjetla s faktorom 0.5 na slici desno

Vrlo jednostavna i brza redukcija curenja svjetla je sve vrijednosti iz intervala $[0, p)$ preslikati u nulu, a interval $[p, 1]$ preslikati na interval $[0, 1]$. Vrijednost p predstavlja faktor redukcije curenja svjetla. Primjer jedne takve funkcije je:

$$\text{Reduction}(v, p) = \begin{cases} 0 & \text{ako je } v < p \\ \frac{v-p}{1-p} & \text{inače} \end{cases}$$

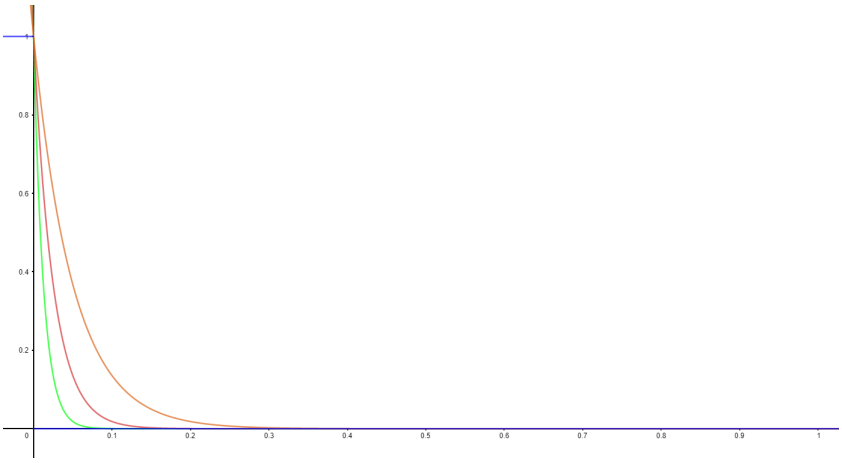
U ovom primjeru preslikvanje $[0, 1] \mapsto [p, 1]$ je linearno, ali takvo preslikavanje može biti bilo koja monotono rastuća neprekidna funkcija na intervalu $[0, 1]$. Općenito takve funkcije mogu davati izlaz izvan intervala $[0, 1]$ pa je takve izlaze potrebno vratiti nazad u interval $[0, 1]$ (npr. korištenjem ugrađene HLSL clamp funkcije). Ovakva redukcija rezultira tamnijim sjenama što je u većini slučajeva prihvatljiva cijena za popravak curenja svjetla.

3.4. Algoritam ESM

Eksponencijalna mapa sjene ili kraće ESM (engl. *Exponential Shadow Maps*) se temelji na opažanju da mapa sjene čuva dubinu najbliže površine što znači da nijedan fragment ne bi trebao imati dubinu manju od one spremljene u mapi sjene ([2]). S pretpostavkom $z \leq \tilde{z}$ funkciju usporedbe (koja je u slučaju obične mape sjene Heavisidova step funkcija) možemo zapisati kao:

$$s(z, \tilde{z}) = \lim_{c \to \infty} \exp(-c(\tilde{z} - z))$$

što se može aproksimirati koristeći neku veliku pozitivnu konstantu c (Slika 3.6). Tada se funkcija usporedbe može faktorizirati:

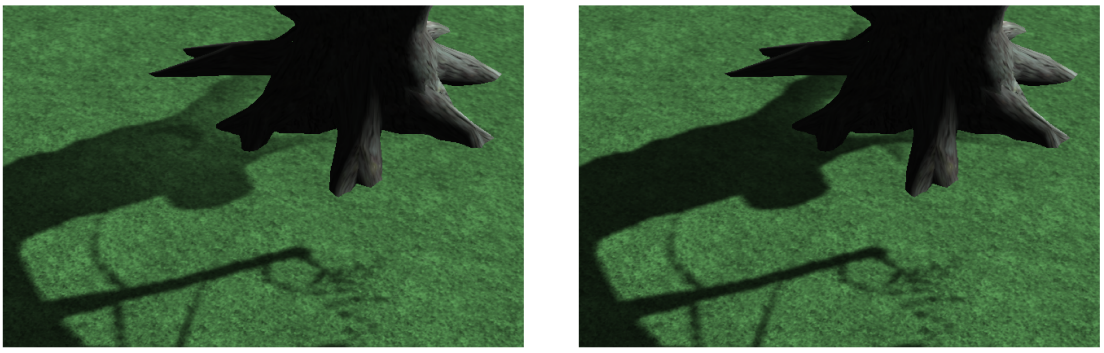


Slika 3.6: Aproksimacija Heavisidove step funkcije na intervalu $[0, 1]$ koristeći eksponencijalnu funkciju.

$$s(z, \tilde{z}) = \exp(-c(\tilde{z} - z)) = \exp(-c\tilde{z}) \exp(cz)$$

Ta konstanta ne može biti bilo koji veliki broj već je ograničena brojem bitova koji se koristi za mapu sjene. Npr. najveći broj koji se može prikazati 32-bitnim brojem s pomičnim zarezom je $\approx 3.4 \cdot 10^{38}$ pa je za takav format mape sjene najveća konstanta koja se može koristiti $\ln(3.4 \cdot 10^{38}) \approx 88.72$ bez da dođe do preljeva. U [2] kao optimalna vrijednost za 32-bitne brojeve s pomičnim zarezom se spominje $c = 80$. Ako vrijednost za c nije dovoljno velika dolazi do curenja svjetlosti. Do curenja svjetlosti i s velikim vrijednostima c dolazi i u slučaju kada su objekt koji baca sjenu i objekt koji ju prima vrlo blizu (Slika 3.7). Tada je $\tilde{z} - z \approx 0$ te $s(z, \tilde{z}) \to 1$. Ako je potrebna vrijednost veća od ≈ 88 (npr. za maloprije spomenuti scenarij) postoji mogućnost zapisivanja linearne dubine u mapu sjene umjesto eksponencijale te zatim filtriranje

provesti u logaritamskom prostoru. Još jedna opcija kod reduciranja curenja svjetla je provesti zatamnjenje sjene ovisno o udaljenosti od objekta koji baca sjenu (engl. *receiver*). Tada je potrebno dodatno čuvati nefiltriranu dubinsku mapu kako bi se mogla odrediti njihova udaljenost te faktor zatamnjenja ([10]).



(a) Artefakti curenja svjetla za $c = 40$ **(b)** Artefakti curenja svjetla za $c = 80$

Slika 3.7: Povećanje faktora c smanjuje curenje svjetla, ali u nekim slučajevima ga ne uklanja potpuno

Primjenom filtra na ovakvu funkciju usporedbe dobije se:

$$\begin{aligned} f_{\text{filter}}(\mathbf{t}, \tilde{z}) &= \sum_{t_i \in \mathcal{K}} k(\mathbf{t}_i - \mathbf{t}) s(z, \tilde{z}) \\ &= \sum_{t_i \in \mathcal{K}} k(\mathbf{t}_i - \mathbf{t}) \exp(-c\tilde{z}) \exp(cz(\mathbf{t}_i)) \\ &= \exp(-c\tilde{z}) \sum_{t_i \in \mathcal{K}} k(\mathbf{t}_i - \mathbf{t}) \exp(cz(\mathbf{t}_i)) \end{aligned}$$

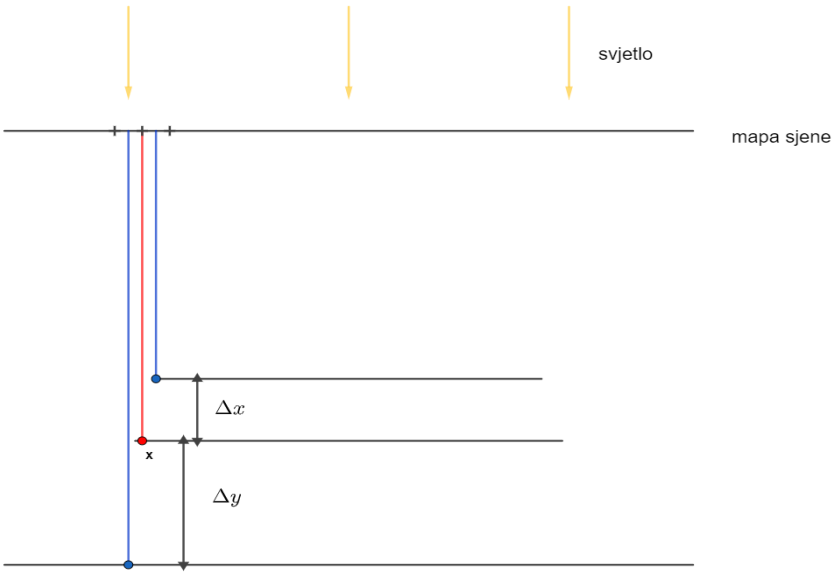
odakle se vidi da faktorizacija funkcije usporedbe $s(z, \tilde{z})$ omogućuje predfiltriranje mape sjene.

Vrijedi spomenuti da je ESM inspiriran konvolucijskom mapom sjena (engl. *Convolution Shadow Maps*, CSM) koja koristi Fourierov red s n članova u aproksimaciji Heavisidove step funkcije. Iako algoritam radi u stvarnom vremenu, zbog velike količine memorije koju zahtjeva i računske složenosti nije prikladan za složenije scene. ESM mijenja aproksimaciju Fourierovim redom s običnom eksponencijalom te u nekim slučajevima daje i bolje rezultate od CSM-a ([2]).

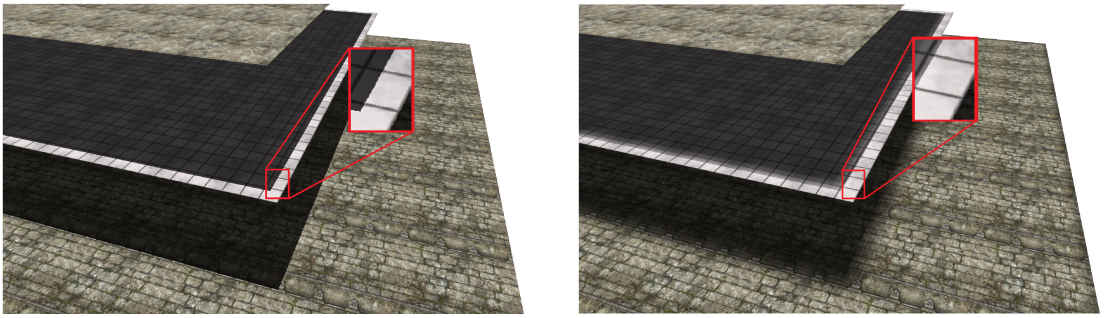
Kršenje pretpostavke

Pretpostavka $z \leq \tilde{z}$ ne vrijedi uvijek, bilo zbog numeričkih nepreciznosti (koje u običnom algoritmu uzrokuju akne sjene) ili filtriranja budući da jezgra filtra može obuhvatiti uzorke $\boldsymbol{t}_i \in \mathcal{K}$ za koje vrijedi $z(\boldsymbol{t}_i) > \tilde{z}$ pa se tada može dogoditi i $z_{\text{avg}} = z > \tilde{z}$

što se zove još problem neplanarnosti (engl. *non-planar problem* [3]). Slika 3.8 prikazuje skicu jednog takvog slučaja, a slika 3.9 jednu takvu scenu. Iako za većinu fragmenata pretpostavka vrijedi, što je veća veličina jezgre $|\mathcal{K}|$ veća je šansa da dođe do kršenja pretpostavke. Ako se uopće ne provodi filtriranje dovoljno je ograničiti dobivenu vrijednost na interval $[0, 1]$. U slučaju filtriranja nije tako jednostavno. Autori u [2] predlažu dvije metode za prepoznavanje fragmenata u kojima dolazi do kršenja pretpostavke. Za takve fragmente se provodi uobičajeni PCF algoritam. Prva metoda naziva klasifikacija temeljena na pragu (engl. *Threshold Classification*) svaki fragment za koji je rezultat $f_{\text{filter}}(\mathbf{t}, \tilde{z}) > 1+\epsilon$ gdje je ϵ prag karakterizira kao "neispravni". Takav pristup nije najprecizniji, ali je jako brz. Prikladan je za jednostavnije scene s manje finih detalja. Druga metoda naziva Z-Max klasifikacija (engl. *Z-Max Classification*) u dodatnu teksturu sprema najveće z-vrijednosti u jezgri filtra za svaki fragment. Ako je $\tilde{z} < z_{\max}$ (drugačije rečeno nejednakost $\tilde{z} < z(\boldsymbol{t}_i)$ vrijedi za barem jedan uzorak filtra), tada se fragment smatra "neispravnim". Z-Max klasifikacija je puno skuplja od klasifikacije temeljene na pragu, ali i preciznija te je potrebno pronaći kompromis između kvalitete i brzine (kao i uvijek).



Slika 3.8: Promatrana točka $\boldsymbol{x}$ ima vidljivost 0.5 koristeći PCF algoritam. Vidljivost izračunata ESM algoritmom je 1 jer vrijedi $z_{\text{avg}} > \tilde{z}$. Iako slika možda tako sugerira, nije nužno da vrijedi $\Delta x < \Delta y$ kako bi došlo do greške zato što se koristi eksponencijalna funkcija. Npr. neka dubina fragmenta $\boldsymbol{x}$ iznosi 0.5, $\Delta x = 0.25$, $\Delta y = 0.25$ i $c = 1$. Tada je $\tilde{z} = \exp(0.5) \approx 1.65$ i $z_{\text{avg}} = \frac{\tilde{z}\exp(-\Delta x)+\tilde{z}\exp(-\Delta y)}{2} > \frac{\tilde{z}(1-\Delta x+1+\Delta y)}{2} = \tilde{z}$ koristeći nejednakost $e^x > 1 + x, x \in \mathbb{R}$. Uvrštavanje vrijednosti $\frac{\exp(0.25)+\exp(0.75)}{2} \approx 1.70$ to potvrđuje. Za usporedbu, aritmetička sredina dubina uzoraka je $z_{\text{avg}} = \tilde{z} + \frac{\Delta y-\Delta x}{2}$.



Slika 3.9: Problem neplanarnosti kod ESM algoritma. Scena je slična onoj na skici 3.8. Slika lijevo prikazuje rezultat dobiven algoritmom mape sjene (u ovom slučaju referentan). Slika desno prikazuje rezultat dobiven algoritmom ESM gdje se mogu primijetiti slikovni elementi čija je vidljivost 1 iako bi trebala biti 0. Korišten je filter 7×7 s relativno velikom razlikom između susjednih uzoraka kako bi problem neplanarnosti bio uočljiviji.

3.5. Algoritam EVSM

EVSM (engl. *Exponential Variance Shadow Map*), predstavljene u [7], koriste pristup VSM-a koji se primjenjuju na mapu sjene u kojoj su spremljene eksponencijale dubina, upravo kao u ESM-u.

Već je rečeno u 3.3 da curenje svjetla uzrokuje rast omjera $t = \frac{\Delta x}{\Delta y}$, a korištenje eksponencijalne funkcije $\exp(kz)$ smanjuje taj omjer jer je udaljenost između objekta A i B u odnosu na udaljenost između B i C u tom slučaju manja. Npr. ako su dubine iznosa $a = 0.25$, $b = 0.5$ i $c = 0.75$ tada su udaljenosti između objekata A i B i objekata B i C jednake, $b - a = c - b = 0.25$ i $t = \frac{\Delta x}{\Delta y} = 1$. Eksponencijalnom funkcijom s npr. $k = 10$ se dobiju nove dubine $a' = \exp(10 \cdot 0.25) \approx 12$, $b' = \exp(10 \cdot 0.5) \approx 148$ i $c' = \exp(10 \cdot 0.75) \approx 1808$ i tada je $b' - a' = 136$, $c' - b' = 1660$ i $t' = \frac{\Delta x'}{\Delta y'} = 0.0819$. Malo formalnije zapisano, vrijedi

$$\frac{t'}{t} = \frac{\frac{\Delta x'}{\Delta y'}}{\frac{\Delta x}{\Delta y}} = \frac{\frac{\exp(kb)-\exp(ka)}{\exp(kc)-\exp(kb)}}{\frac{b-a}{c-b}} = \frac{(c-b)(\exp(kb)-\exp(ka))}{(b-a)(\exp(kc)-\exp(kb))} < 1, \quad a \leq b \leq c$$

Veća konstanta k dodatno smanjuje taj omjer. Npr. u gornjem slučaju za $k = 80$ je $t' = 2.06 \cdot 10^{-9}$.

Lauritzen u [7] predlaže korištenje i $-\exp(-kz)$ kako bi se izbjegao problem neplanarnosti u ESM-u. Ova funkcija djeluje suprotno, pa je udaljenost između objekta B i C relativno manja što je i poželjno jer su tada objekti koji primaju sjenu (B i C) relativno bliži čime se smanjuje utjecaj neplanarnosti. Dvije funkcije $\exp(kz)$ i $-\exp(-kz)$ se

mogu koristiti zajedno što znači da se koriste sva četiri kanala mape sjene u koji se redom spremaju - $\exp(kz)$, $\exp(2kz)$,$-\exp(-kz)$,$-\exp(-2kz)$. Na taj način dobijemo dvije gornje međe na vidljivost fragmenta (s Čebiševljevom nejednakošću) te uzimanje manje od te dvije daje dobru aproksimaciju vidljivosti. Artefakti se pojavljuju na mjestima gdje i VSM i ESM imaju artefakti, a s većim k se ublažavaju. Cijena boljih sjena s manje artefakata je povećano korištenje memorije s obzirom na to da se koriste sva četiri kanala mape sjene. U slučaju EVSM-a najveća vrijednost za k dvostruko manja nego u slučaju ESM-a, tj. ≈ 44.36.
Slika 3.10 uspoređuje rezultate dobivene s VSM, ESM i EVSM s referentnim PCF algoritmom.

3.6. Algoritam PCSS

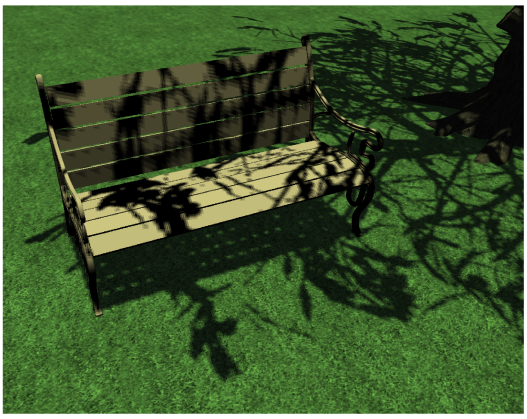
Problem kod PCF algoritma (a i kod svih ostalih dosad obrađenih algoritama) je što je širina jezgre filtriranja, odnosno širina penumbre sjene uvijek ista. To je u nekim slučajevima prihvatljivo, ali realistične sjene su oštrije pri kontaktu objekata i mekše kad su ti objekti udaljeniji. PCSS ili engl. *Percentage-Closer Soft Shadows* je algoritam koji pokušava postići upravo taj efekt ([6]). Temelji se na opažanju da su sjene mekše što je veličina/širina jezgre filtra veća. PCSS algoritam se može interpretirati kao PCF algoritam s varijabilnom širinom jezgre filtra. Može se rastaviti na tri koraka:

1. **Računanje zaklonjenosti** (engl. *Blocker search*) - pretražuje se mapa sjene u nekoj regiji $\mathcal{R}$ te se izračuna aritmetička sredina svih dubina koje su bliže izvoru svjetlosti nego promatrana točka. Širina regije $\mathcal{R}$ ovisi o veličini svjetla te udaljenosti promatrane točke od izvora. Iz sličnih trokuta (slika 3.12 lijevo) se za širinu pretraživanja dobije:

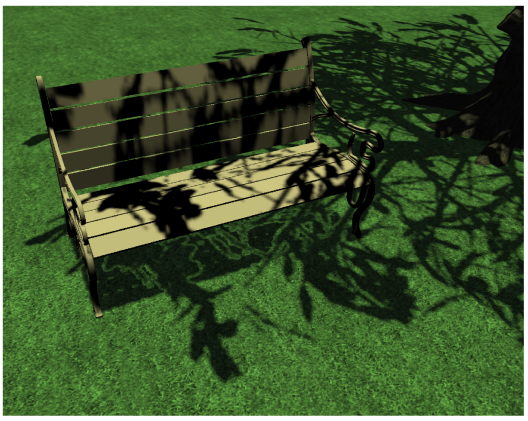
$$\frac{w_{\text{light}}}{\tilde{z}} = \frac{w_{\text{search}}}{\tilde{z} - z_{\text{near}}} \Rightarrow w_{\text{search}} = \frac{\tilde{z} - z_{\text{near}}}{\tilde{z}} \cdot w_{\text{light}}$$

   Naravno, ako ne postoji nijedna dubina koja je bliža izvoru od promatrane, točka je maksimalno osvijetljena te se ostali koraci ne moraju provoditi.

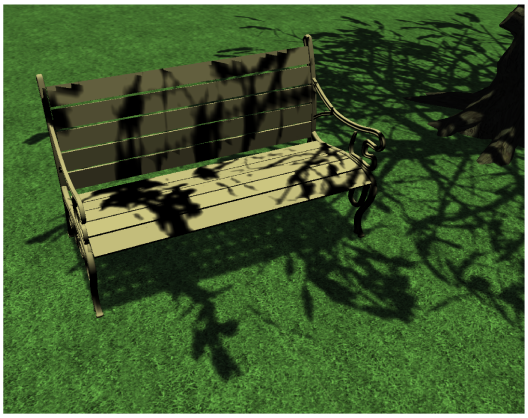
2. **Procjena širine penumbre** (engl. *Penumbra-width estimation*) - kako bi se odredila širina jezgre filtra, PCSS koristi pretpostavku da postoji samo jedan objekt koji zaklanja svjetlo i koji je paralelan s izvorom svjetlosti (koje se ovdje smatra površinsko svjetlo, engl. *Area light*, ali i ostalim tipovima svjetla možemo pridružiti vrijednost koja predstavlja njihovu veličine te time određuje



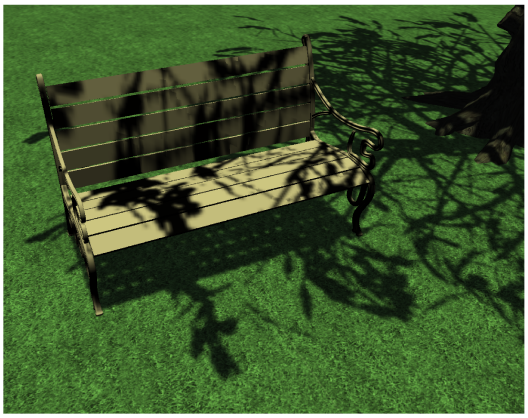
(a) Rezultat dobijen PCF algoritmom. Služi kao referentan primjer kako bi se lakše uočili artefakti i curenje svjetla kod drugih algoritama



(b) Rezultat dobijen VSM algoritmom. Curenje svjetla ispod klup nije moguće spriječiti bez gubitka detalja sjene uslijed njihovog zadebljanja

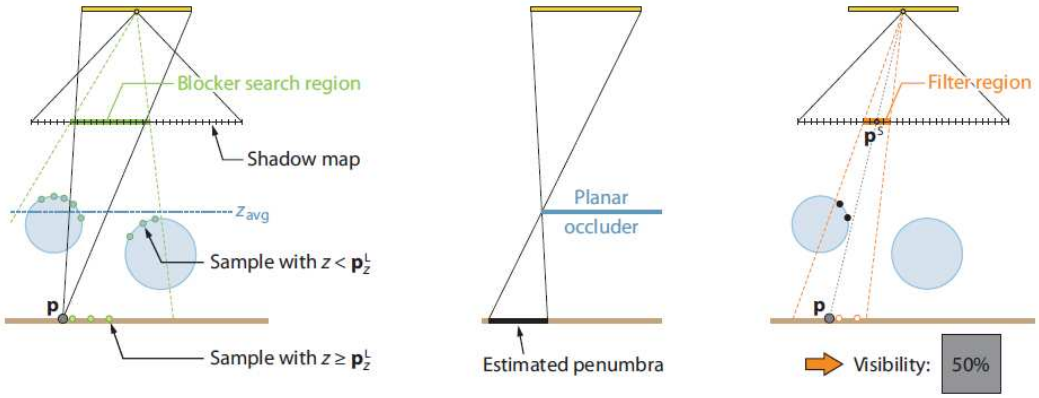


(c) Rezultat dobiven ESM algoritmom. Korišten je eksponent $c = 85$. I dalje se vidi blago curenje svjetla ispod klupe.



(d) Rezultat dobiven EVSM algoritmom. Korišten je eksponent $c = 40$. Nije bilo potrebno koristiti faktor redukcije.

Slika 3.10: Usporedba obrađenih metoda. Veličina mape sjene je 2048×2048, veličina filtra je 3×3



Slika 3.11: Tri koraka u algoritmu PCSS (Slika iz [5])

mekoću sjene) te leži na dubinu z_{avg}, izračunatoj u prošlom koraku. Vidi sliku 3.12. Iz sličnih trokuta se za širinu penumbre dobije:

$$w_{\text{penumbra}} = \frac{\tilde{z} - z_{\text{avg}}}{z_{\text{avg}}} \cdot w_{\text{light}}$$

3. **Filtriranje** - provodi se uobičajeni PCF sa širinom filtra proporcionalna izračunatoj širinom penumbre. Jedna opcija je projicirati širinu penumbre na mapu sjene (slika 3.12 desno). Iz sličnih trokuta se izračuna:

$$\frac{w_f}{z_{\text{near}}} = \frac{w_{\text{penumbra}}}{z} \Rightarrow w_f = \frac{z_{\text{near}}}{z} \cdot w_{\text{penumbra}}$$

Kako bi se izbjegao alias-efekt kada je $w_f \approx 0$ može se koristiti neka minimalna vrijednost $w_{f\text{min}}$ i $w_f = \max(w_f, w_{f\text{min}})$.

Nedostatci

Iako PCSS daje vizualno prihvatljive rezultate, zbog pretpostavki koje koristi sjene koje stvara najčešće nisu potpuno točne. Npr. na slici 3.12 lijevo, za sve uzorke na desnom krugu (a neki i na lijevom) se smatra da blokiraju svjetlost do promatrane točke što nije istina i može dovesti značajne pogreške u računanju z_{avg} što se kasnije propagira na ostale korake te dovodi do neispravno izračunate vidljivosti.
Veliki broj uzorkovanja teksture zbog računanja zaklonjenosti i filtriranja definitivno utječe na performanse pa se u praksi regija pretraživanja kod računanja zaklonjenosti $\mathcal{R}$ ne pretražuje temeljito već s ograničenim brojem uzoraka (16 do 25 uzoraka je obično dovoljno [5]). Predfiltriranje u PCSS-u nije moguće jer širina filtera ovisi o promatranoj točki. Postoje algoritmi koji postižu sličan efekt kao i PCSS i temelje

(a) PCF algoritam

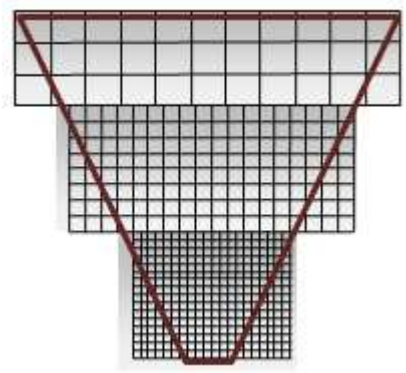
(b) PCSS algoritam

Slika 3.12: Rezultati dobiveni PCF i PCSS algoritmom. Kontaktne sjene su oštrije u slučaju PCSS-a, a mekoća udaljenijih sjena se može podesiti parametrom mekoće (engl. softness) koji se može interpretirati kao veličina svjetla

se na algoritmima koji omogućuju predfiltriranje poput VSSM (engl. *Variance Soft Shadow Mapping*) koji se temelji na VSM-u ([3]).

4. Kaskadne mape sjena

Kaskadne mape sjena su najistaknutiji algoritam u borbi protiv perspektivnog alias-efekta te je zbog svojih rezultata često korišten u industriji ([9]). Koncept je jednostavan, različiti dijelovi volumena pogleda kamere zahtijevaju različitu gustoću uzoraka mape sjene. Kaskadne mape sjene to postižu podjelom volumena pogleda kamere koji je krnja piramida na više manjih krnjih piramida. Mapa sjene se iscrtava za svaku manju krnju piramidu, a fragment uzorkuje onu mapu s najprikladnijom gustoćom uzoraka. Slika pokazuje gustoću uzoraka mape sjena za tri kaskade. Filtriranje kaskadnih mapa sjena se obavlja kao i prije, samo što se, u slučaju predfiltriranja, taj postupak obavlja za svaku kaskadu. Ideja kaskadnih sjena je prvi put predstavljena u [13]. Prvi

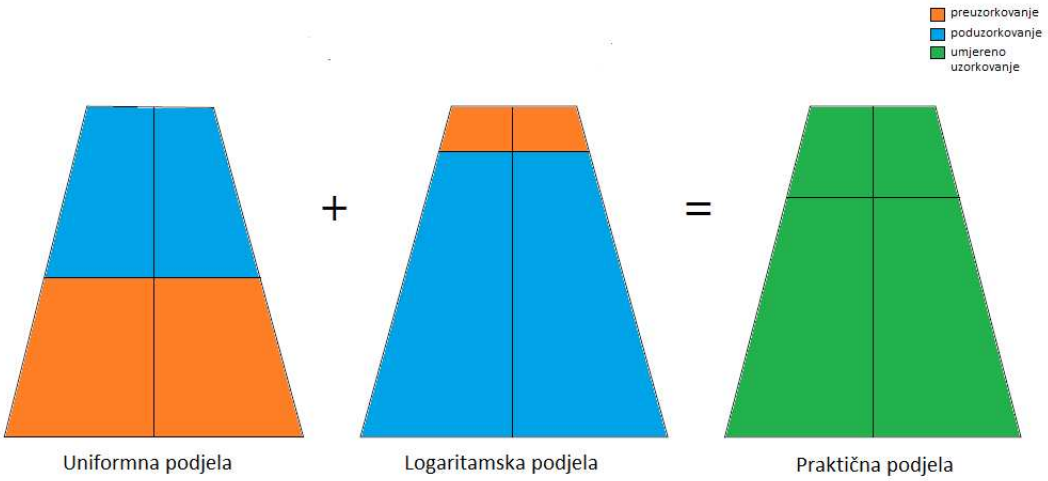


Slika 4.1: Gustoća uzoraka mapa sjena u CSM-u ([9])

korak CSM-a je odabir podjele volumena pogleda kamere.

4.1. Odabir podjele

Prije samog iscrtavanja mapa sjena prvo je potrebno odrediti na koji način se vrši podjela krnje piramide kamere. Najčešće se razmatraju tri vrste podjele: uniformna, logaritamska i praktična ([13]). Ako se koristi N kaskada, tada se njihove lokacije duž z-osi označavaju s $C_i, 0 \leq i < N$ i vrijedi $C_0 = n$ i $C_N = f$. Početak jedne kaskade



Slika 4.2: Praktična podjela je dobivena kao težinska sredina uniformne i logaritamske podjele

označava kraj prethodne pa je dovoljno odrediti samo početke. C_N ne označava početak nijedne kaskade već samo kraj zadnje (s n i f se označavaju najmanja i najveća udaljenost krnje piramide po z-osi - engl. *near* i *far*).

Kako bi se donekle mogla odrediti efikasnost podjele, koristi se formula iz 2.3 za perspektivnu grešku poduzorkovanja:

$$\tilde{m} = \frac{n}{z}\frac{dz}{ds}$$

Uniformna podjela

Uniformna podjela je najjednostavnija podjela u kojoj su udaljenosti između susjednih podjela jednake. Formula je u tom slučaju:

$$C_i^{\text{uniform}} = n + i \cdot \frac{f-n}{N}$$

i udaljenost između susjednih podjela je:

$$C_{i+1}^{\text{uniform}} - C_i^{\text{uniform}} = \frac{f-n}{N}, \quad \forall i = 0, 1, \ldots, N-1$$

U uniformnoj podjeli vrijedi

$$z = n + (f-n)s \Rightarrow \frac{dz}{ds} = f - n \Rightarrow \tilde{m} = \frac{n(f-n)}{z}$$

To znači da greška poduzorkovanja u uniformnoj parametrizaciji raste hiperbolički kada se objekt približava očištu. Uniformna podjela je teorijski najgora podjela ([13]).

Logaritamska podjela

Optimalna distribucija greške poduzorkovanja je konstantna, tj. $\frac{dz}{z\,ds} = \rho, \rho \in \mathbb{R}$, odnosno

$$ds = \frac{1}{\rho}\frac{dz}{z}$$
$$s = \int_0^s ds = \frac{1}{\rho}\int_n^z \frac{dz}{z} = \frac{1}{\rho}\ln\left(\frac{z}{n}\right)$$

Ovdje je također pretpostavljeno (kao i u analizi greške) da mapa sjena sadrži samo vidljivi dio scene što ne vrijedi u praksi. Za s vrijedi $s \in [0, 1]$ pa se lako dobije $\rho = \ln\left(\frac{f}{n}\right)$ i

$$s(z) = \frac{\ln\left(\frac{z}{n}\right)}{\ln\left(\frac{f}{n}\right)}$$

Kako bi dobili lokacije kaskada diskretizira se u $z = C_i^{\log}$:

$$s_i = s_i(C_i^{\log}) = \frac{\ln\left(\frac{C_i^{\log}}{n}\right)}{\ln\left(\frac{f}{n}\right)}$$
$$C_i^{\log} = n\left(\frac{f}{n}\right)^{s_i}$$

S obzirom da logaritamska podjela pokušava postići konstantnu distribuciju greške tada $s_i = \frac{i}{N}$ pa je konačna vrijednost za lokacije kaskada:

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

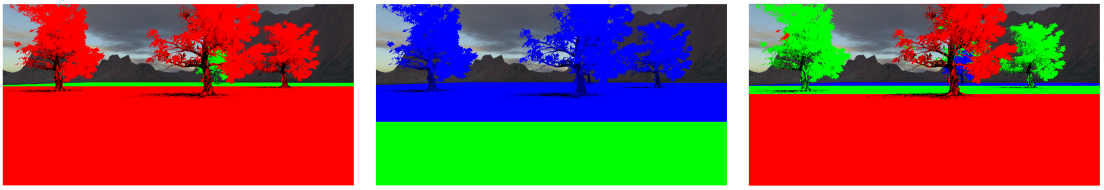
Međutim ovo je samo gruba aproksimacija jer broj N u teoriji teži u beskonačnost, a u praksi N je obično između 3 i 6. Ova podjela nije praktična zato što su dužine kaskada blizu očišta premale što znači da se puno memorije troši na dio scene u kojem ima malo objekata.

Praktična podjela

Ni logaritamska, a pogotovo uniformna ne mogu postići prikladno uzorkovanje za cijeli volumen pogleda. U [13] predlažu težinsku sredinu te dvije podjele. Tada se C_i računa na idući način:

$$\begin{aligned} C_i &= \lambda C_i^{\log} + (1-\lambda)C_i^{\text{uniform}} \\ &= \lambda n\left(\frac{f}{n}\right)^{\frac{i}{N}} + (1-\lambda)\left(n + i\cdot\frac{f-n}{N}\right) \end{aligned}$$

Visualizacija kaskada za tri spomenute podjele se može vidjeti na slici 4.3.



(a) Uniformna podjela **(b)** Logaritamska podjela **(c)** Praktična podjela s $\lambda = 0.5$

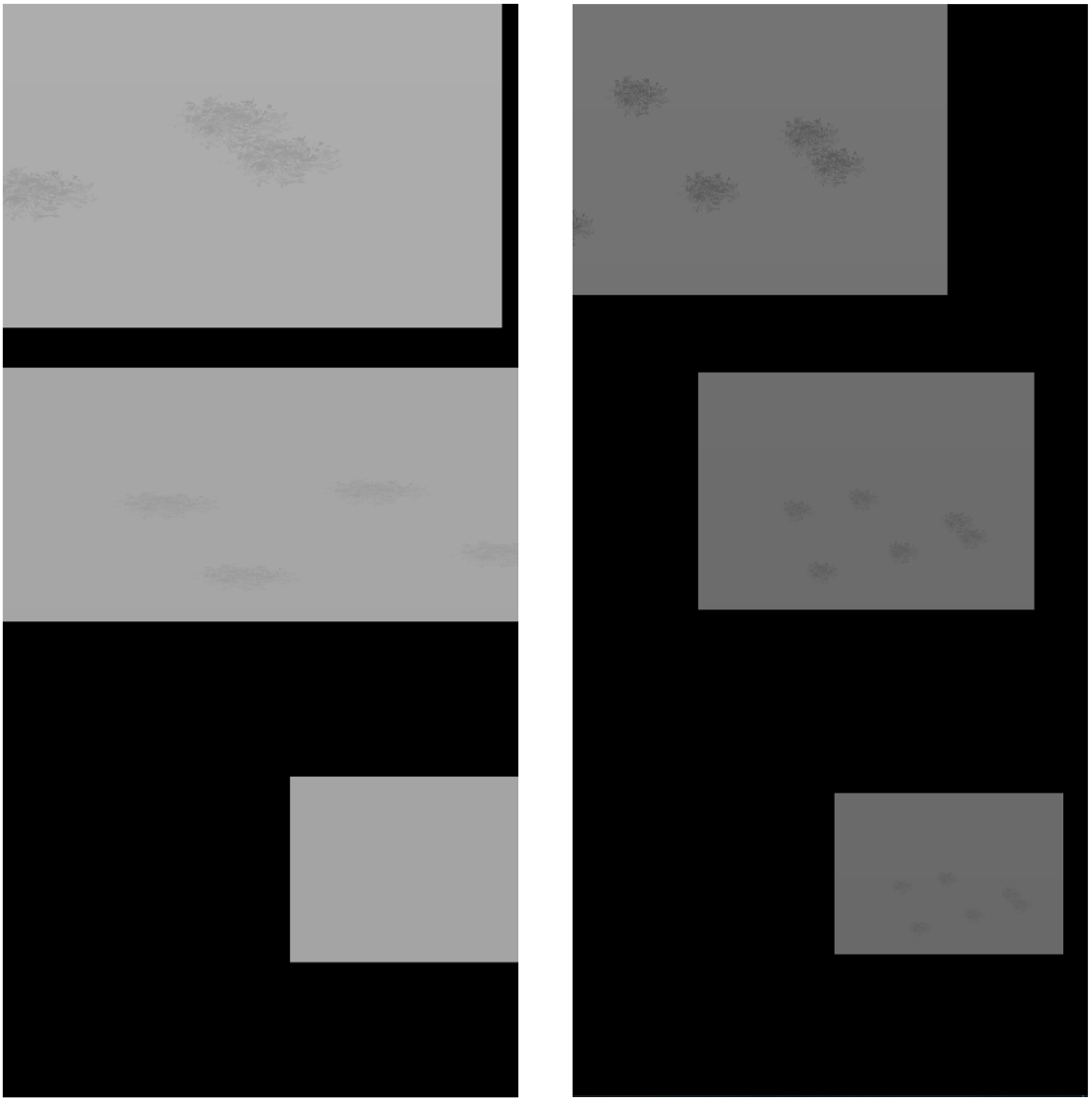
Slika 4.3: Visualizacija kaskada gdje je $N = 3$. Najbliža kaskada je obojana crvenom, najudaljenija plavom, a srednja zelenom.

4.2. Stabilne kaskadne mape sjena

Konstrukcija matrice pogleda i projekcije za svaku kaskadu je objašnjena u 2.2, ali ovdje će se detaljnije razmotriti dvije mogućnosti kod konstrukcije matrice projekcije te njihov utjecaj na stabilnost mapa sjena. Koraci objašnjeni u nastavku se obavljaju za svaku kaskadu te pretpostavljaju usmjereno svjetlo.

Prvi korak je pronaći vrhove manje krnje piramide za tu kaskadu. Kako bi odredili poziciju svjetla izračuna se središte krnje piramide kao aritmetička sredina vrhova. Pozicija svjetla se izračuna kao u 2.2. Vrijedi napomenuti da svaka kaskada ima drugačiju poziciju izvora svjetlosti. Sada, znajući poziciju izvora svjetlosti, se lako odredi matrica pogleda.

Kod računanja matrice projekcije postoje dvije opcije, izračunati parametre projekcije na temelju omeđujućeg kvadra ili na temelju omeđujuće sfere. Ako se odabere opcija s omeđujućim kvadrom dolazi do neželjene pojave koja se naziva i plivanje sjena koja se manifestira pomicanjem rubova sjena (engl. *shadow swimming*, *shadow shimmering*). Dva su uzroka ove pojave. Prvi je rotacija kamere. Rotacijom kamere dolazi do promjene u veličini omeđujućeg kvadra što znači i promjena veličine volumena pogleda, odnosno preslikavanje između elemenata mape sjene i slikovnih elemenata je drugačije svaki put kad se kamera rotira. Rješenje ovog uzroka je koristiti omeđujuću sferu umjesto omeđujućeg kvadra koja je invarijantna na rotacije čime veličina volumena pogleda ostaje uvijek ista. Kako bi mogli izračunati volumen pogleda koji za ortografsku projekciju ima oblik kvadra dovoljno je naći omeđujući kvadar omeđujuće sfere. Drugi uzrok je translacija kamere što uzrokuje i translaciju pozicije svjetla za svaku kaskadu. U tom slučaju geometrija scene se pomiče u podtekselnim inkrementima (engl. *sub-texel increments*) u mapi sjene što uzrokuje plivanje sjena. Rješenje je poziciju svjetla pomicati u cijelim tekselnim inkrementima (engl. *whole-texel increments*). Tako dobivene kaskadne mape sjena se nazivaju stabilne kaskadne mape sjena. Cijena stabilnosti je gubitak preciznosti mapa sjena što se može vidjeti na slici 4.4.



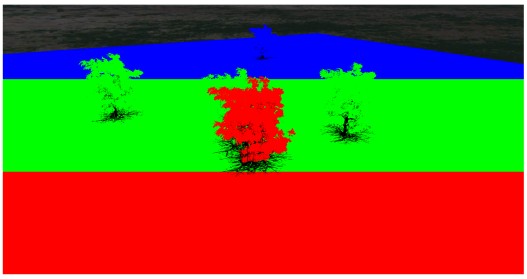
Slika 4.4: Gubitak preciznosti mapa sjena kod stabilizacije. Slika lijevo prikazuje mape sjene bez stabilizacije, a slika desno sa stabilizacijom.

4.3. Odabir kaskade

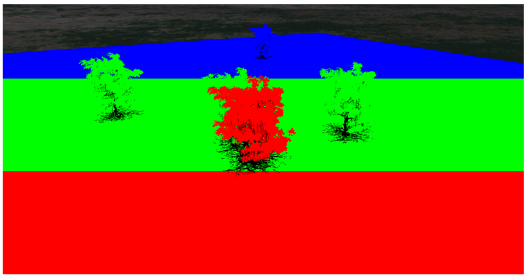
Dvije najčešće metode odabira kaskade su ([9]):

1. Odabir kaskade temeljen na intervalu (engl. *Interval-Based Cascade Selection*). Kaskadne mape sjene s ovakvim odabirom kaskada se nazivaju još i engl. *Parallel Split Shadow Maps* (PSSM). Odabrana kaskada ovisi o fragmentovoj z-koordinati u prostoru kamere i određenim lokacijama kaskada C_i. Bira se prva kaskada za koju vrijedi $z_{\text{view}} < C_i$.

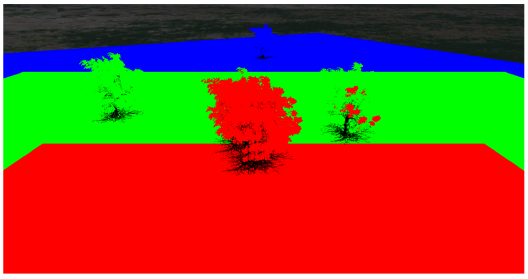
2. Odabir kaskade temeljen na mapi (engl. *Map-Based Cascade Selection*). Bira se mapa najveće preciznosti u kojoj je zapisana dubina promatranog fragmenta.



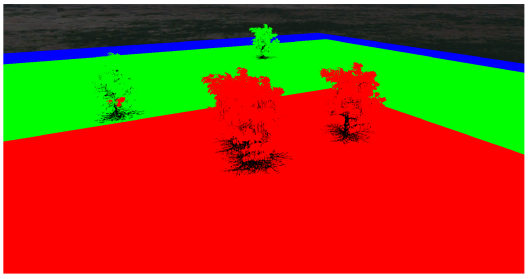
(a) Odabir kaskada temeljen na intervalu bez stabilizacije



(b) Odabir kaskada temeljen na intervalu sa stabilizacijom



(c) Odabir kaskada temeljen na mapi bez stabilizacije



(d) Odabir kaskada temeljen na mapi sa stabilizacijom

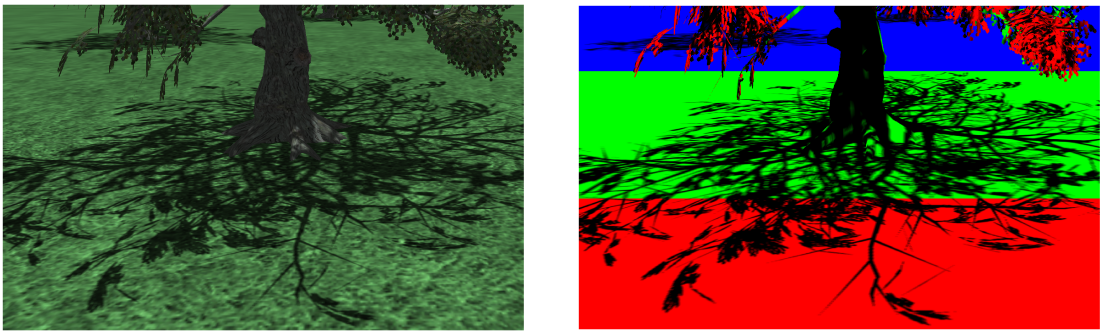
Slika 4.5: Visualizacija kaskada ovisno o odabiru kaskade te stabilizaciji. U slučaju odabira kaskada temeljenog na intervalu ne postoji razlika između stabiliziranih i nestabliziranih mapa sjena što ima smisla jer odabir ovisi samo o z-koordinati fragmenta u prostoru kamere. Korištena je praktična podjela s $\lambda = 0.6$.

> Izračunaju se koordinate za uzorkovanje teksture te se provjeri jesu li obje u intervalu $[0, 1]$. Ako jesu, ta mapa sadrži dubinu promatranog fragmenta.

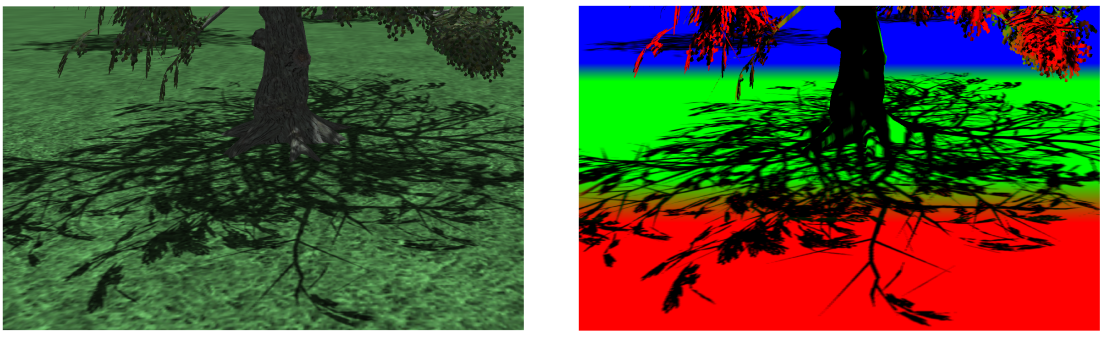
Na slici 4.5 se može vidjeti visualizacija kaskada ovisno o načinu odabira kaskade i stabilizaciji.

4.4. Prijelaz između kaskada

Na mjestima prijelaza između kaskada, pogotovo kod onih kaskada bliže očištu, može doći do vidljive razlike u izgledu sjena zbog nagle promjene rezolucije mape sjene. Ta razlika je još očitija ako se koristi filtriranje mapa sjena poput PCF-a ili VSM-a. Slika 4.6 pokazuje jednu takvu situaciju. Rješenje je stvoriti pojas oko mjesta prijelaza gdje se izračuna vidljivost fragmenta za obje kaskade te se, ovisno o poziciji fragmenta u prostoru kamere, konačna vidljivost izračuna linearnom interpolacijom između dvije izračunate vidljivosti ([9]). Vidi sliku 4.7.



Slika 4.6: Nagla promjena u rezoluciji mape uzrokuje uočljive razlike na prijelazima između kaskada



Slika 4.7: Linearnom interpolacijom se ublažavaju uočljive razlike na prijelazima kaskada. Veličina pojasa je ± 10 %.

5. Implementacija

U sklopu rada je izrađena programska implementacija koristeći razvojno okruženje Microsoft Visual Studio 2019, programski jezik C++, grafičko programsko sučelje DirectX 11, jezik za sjenčanje HLSL, knjižnica assimp za učitavanje 3D modela te knjižnica ImGui za korisničko sučelje. Za stvaranje prozora te obradu korisničkog unosa poput tipkovnice i miša koristi se Win32 API.
Za potrebe implementacije napravljen je vlastiti radni okvir koji sadrži tanke omotače oko DirectX objekata, sustav za stvaranje prozora u Win32 API-ju te razne pomoćne funkcije i klase. Za apstrakciju objekata (poput 3D modela, svjetla i slično, ne DirectX objekata), njihovih svojstava te sustava koji djeluju nad njima koristi se jednostavna vlastita implementacija ECS arhitekture (engl. *Entity Component System*). Objekti ili entiteti u takvoj arhitekturi mogu biti i obični brojevi (A.1). Npr. sustav za iscrtavanje zahtjeva da su objekti koje on iscrtava vidljivi kako bi se smanjio broj poziva prema GPU (u ECS-u to znači da ima komponentu koja označava da je objekt u tom trenutku vidljiv), a vidljivost određuje sustav posebno zadužen za taj zadatak. Da bi taj sustav odredio vidljivost objekt mora imati komponentu AABB (engl. *Aligned Axis Bounding Box*) kako bi odredio postoje li preklapanja između krnje piramide kamere i omeđujućeg kvadra objekta (A.2). Postupak stvaranja objekta u ovakvoj arhitekturi, npr. svjetla se može vidjeti u A.3.

Implementacija sadrži sve algoritme obrađene u ovom radu te je podijeljena na dvije aplikacije. Prva aplikacija sadrži algoritme iz poglavlja 3, dok druga aplikacija sadrži algoritme iz poglavlja 4. Zanimljiviji isječci kodova, bio to C++ ili HLSL kod, se mogu vidjeti u dodatku A.

5.1. Filtrirane oštre sjene

Slika 5.2 prikazuje korisničko sučelje u aplikaciji za filtrirane oštre sjene. Svaki prozor korisničkog sučelja se može minimizirati. Osim promjene smjera i boje usmjerenog

svjetla, korisnik u prozoru "Shadow Technique" može birati između šest algoritama:

1. PCF. U slučaju kada se stavi opcija "No filtering", to zapravo postane osnovna verzija algoritma mape sjene (namjerno se ne koristi hardverski 2×2 PCF baš zbog toga)
2. VSM.
3. ESM.
4. EVSM.
5. MSM. MSM ili mapa sjene temeljena na momentima (engl. *Moment Shadow Maps*) nije obrađena u radu, ali je dodan u implementaciju s obzirom da su autori algoritma dali HLSL kod koji je olakšao integraciju algoritma u radni okvir.
6. PCSS.

Slika 5.1: Korisničko sučelje u aplikaciji za filtrirane oštre sjene

Ovisno o odabranom algoritmu, prozor s postavkama ("Shadow settings") se može promijeniti. Ono što uvijek ostaje isto je:

- Odabir veličine mape sjene. Moguće opcije su 512×512, 1024×1024, 2048×2048, 4096×4096 i 8192×8192
- Odabir metode filtriranja. Moguće opcije su bez filtriranja, 3×3, 5×5, 7×7 (PCF ima dodatnu opciju Poissonovo uzorkovanje)

– Mekoća sjene. Taj parametar zapravo određuje udaljenost između uzoraka u teksturi, tj. širinu jezgre filtra

– Postavke za bias. Iako nekim algoritmima poput VSM-a nije potreban nalazi se u postavkama svih algoritama

– Ostale opcije poput iscrtavanja mape sjene u desnom donjem kutu prozora i podrška za transparentne objekte

Dodatne opcije koje nude određeni algoritmi su:

1. VSM ima opciju biranja faktora redukcije svjetla iz intervala $[0, 0.95]$
2. ESM ima opciju biranja eksponenta iz intervala $[1, 88]$
3. EVSM ima opciju biranja eksponenta iz intervala $[1, 44]$ i biranja faktora redukcije svjetla iz intervala $[0, 0.95]$
4. MSM također ima jedan parametar za redukciju curenja svjetla koji se razlikuje od onog iz VSM algoritma

Mjerenja

Kao mjeru brzine uzima se broj sličica po sekundi ili FPS (engl. *Frames Per Second*). Prvo mjerenje uspoređuje brzinu algoritama filtriranih mapa sjena ovisno o veličini jezgre filtra.

	PCF	VSM	ESM	EVSM
1×1	386	273	277	199
3×3	277	229	232	124
5×5	141	192	197	113
7×7	86	177	180	97

Veličina prozora 1920×1080, a mape sjene 1024×1024. Pad FPS-a kod PCF-a je puno veći nego kod ostalih algoritama što je i očekivano jer je složenost filtriranja u tom slučaju kvadratna, a za ostale je linearna.

Drugo mjerenje uspoređuje brzinu algoritama filtriranih mapa sjena ovisno o veličini mape sjene.

	PCF	VSM	ESM	EVSM
512×512	281	284	289	210
1024×1024	277	225	230	121
2048×2048	275	122	124	48

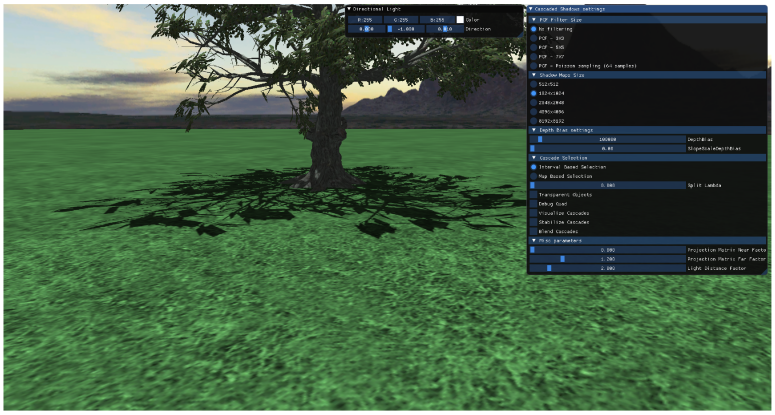
Veličina prozora 1920×1080, a filtriranje je 3×3. Razlike u FPS-u kod algoritma PCF-a gotovo da i nema. Drugi algoritmi provode filtriranje direktno nad mapom sjena te dolazi do značajne razlike u FPS-u. Iako je složenost takvog filtriranja linearna, ta prednost se "istopila" koristeći veću mapu sjene. (PCF filtriranje je kvadratne složenosti, ali to se obavlja u prostoru prozora. Obrnute rezultate bi dobili mjenjajući veličinu prozora držeći veličinu mape sjene fiksnom.)

5.2. Kaskadne mape sjena

Slika 5.2 prikazuje korisničko sučelje u aplikaciji za kaskadne mape sjena. Svaki prozor korisničkog sučelja se može minimizirati. Osim postavki vezanih za svjetlo (koje su iste kao i u prvoj aplikaciji), korisnik može birati između sljedećih opcija:

- Odabir veličine mape sjene. Moguće opcije su 512×512, 1024×1024, 2048×2048, 4096×4096 i 8192×8192
- Odabir metode filtriranja. Moguće opcije su bez filtriranja, 3×3, 5×5, 7×7 i Poissonovo uzorkovanje sa 64 uzorka. Radi jednostavnosti je samo PCF algoritam podržan.
- Postavke za bias.
- Odabir kaskade. Podržana su oba obrađena načina, temeljen na intervalu i temeljen na mapi.
- Podrška za transparentne objekte
- Iscrtavanje kaskada na desnoj strani ekrana (tada kaskade zauzimaju trećinu prozora). Broj kaskada je uvijek tri.
- Vizualizacija kaskada. Najbliža kaskada je obojana crveno, srednja zeleno, a najdalja plavo.
- Stabilizacija kaskada.
- Interpolacija između kaskada. Ova opcija je podržana samo za odabir kaskada temeljen na intervalu. Veličina pojasa je uvijek ± 10 %.
- Postavke vezane za projekcijsku matricu i matricu pogleda. Za neke scene je

potrebno postaviti neke parametre kako bi sva geometrija koja baca sjenu u vidljiv dio scene bila u volumenu pogleda svjetla.



Slika 5.2: Korisničko sučelje u aplikaciji za kaskadne mape sjena

Mjerenja

Prvo mjerenje uspoređuje brzinu algoritma kaskadne mape sjena ovisno o veličini mapa sjena. Sve mape sjene, tj. kaskade su iste veličine. Drugo mjerenje uspore-

Veličina mapa sjena	Broj FPS-a
512×512	158
1024×1024	146
2048×2048	103

Brzina algoritma ovisno o veličini mapa sjena bez filtriranja s odabirom kaskade temeljenom na intervalu. Veličina prozora je 1920×1080.

đuje brzinu algoritma kaskadne mape sjena ovisno o veličini jezgre filtra.

Veličina jezgre filtra	Broj FPS-a
1×1	146
3×3	121
5×5	86
7×7	61

Brzina algoritma ovisno o veličini jezgre filtra s odabirom kaskade temeljenom na intervalu. Veličina prozora je 1920×1080, a veličina mape sjene je 1024×1024.

Interpolacija između kaskada na testnoj sceni uzrokuje blagi pad sa 146 na 137 FPS-a. Mjenjanje odabira kaskada uzrokuje još blaži pad sa 146 na 143 FPS-a.

6. Zaključak

U ovom radu su predstavljeni neki od najkorištenijih algoritama za prikaz sjena temeljeni na mapi sjene. Fokus je bio na trenutno najkorištenijim pristupima u industriji poput filtriranih mapa sjena i kaskadnih mapa sjena. Filtriranjem rubovi sjena postanu mekši i prirodniji od nazubljenih rubova kakvi su dobije bez filtriranja. Za svaki od njih je napisana implementacija kako bi se mogli usporediti i prokomentirati rezultati te svi djeluju u stvarnom vremenu. Algoritam PCF je najprecizniji te se pri usporedbama koristi kao referentni primjer, ali zbog nemogućnosti predfiltriranja u nekim slučajevima postaje relativno skup. Ako je veličina mape sjene puno veća od veličine prozora ili je broj uzoraka malen, prednost predfiltriranja se gubi te je u tom slučaju bolje koristiti PCF. Algoritmi VSM i ESM omogućuju predfiltriranje, ali cijena toga su nepoželjni artefakti poput curenja svjetla kojih se nije moguće uvijek riješiti. EVSM koristi prednosti oba ta pristupa te curenje svjetla svodi na minimum uz cijenu povećane memorije. Obrađen je i predstavnik algoritama za postizanje mekih sjena PCSS koji se može interpretirati kao PCF algoritam s varijabilnom širinom jezgre filtra. Osim uzorkovanja mape sjene u procesu filtriranja, i drugi dijelovi ovog algoritma zahtijevaju određeni broj uzorkovanja što ga čini nešto sporijim od PCF-a. Kaskadne mape sjene su prikladne za velike otvorene prostore gdje ostali obrađeni algoritmi daju lošije rezultate zbog velikog prostora koji mapa sjene mora pokriti čime dolazi do velikog gubitka rezolucije. Koristi se više mapa sjena s različitom gustoćom uzoraka te se ovisno o udaljenosti od očišta uzorkuje određena mapa. Tako dolazi do smanjenja perspektivne greške poduzorkovanja, a posljedično tome i bolje kvalitete sjena. Kaskadne mape sjena se mogu koristiti u kombinaciji s filtriranim mapa sjena te tako iskoristiti prednosti oba pristupa.

LITERATURA

[1] Tomas Akenine-Moller, Eric Haines i Naty Hoffman. *Real-Time Rendering*. 2008.

[2] Thomas Annen i dr. "Exponential Shadow Maps". *Proceedings of Graphics Interface 2008*. GI '08. Windsor, Ontario, Canada: Canadian Information Processing Society, 2008., 155–161.

[3] Zhao Dong i Baoguang Yang. "Variance Soft Shadow Mapping". *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. I3D '10. Washington, D.C.: Association for Computing Machinery, 2010.

[4] William Donnelly i Andrew Lauritzen. "Variance Shadow Maps". *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*. I3D '06. Redwood City, California: Association for Computing Machinery, 2006., 161–165.

[5] Elmar Eisemann i dr. *Real-Time Shadows*. 2011.

[6] Randima Fernando. "Percentage-Closer Soft Shadows". *ACM SIGGRAPH 2005 Sketches*. SIGGRAPH '05. Los Angeles, California: Association for Computing Machinery, 2005., 35–es.

[7] Andrew Lauritzen i Michael McCool. "Layered Variance Shadow Maps". *Proceedings of Graphics Interface 2008*. GI '08. Windsor, Ontario, Canada: Canadian Information Processing Society, 2008., 139–146.

[8] D. Brandon Lloyd i dr. "Logarithmic Perspective Shadow Maps". Sv. 27. 4. New York, NY, USA: Association for Computing Machinery, studeni 2008.

[9] Microsoft. *Cascaded Shadow Maps*. `https://docs.microsoft.com/en-us/windows/win32/dxtecharts/cascaded-shadow-maps`. Accessed on 2020-06-30. 2018.

[10] Natalya Tatarchuk. "Advances in Real-Time Rendering in 3D Graphics and Games I". *ACM SIGGRAPH 2009 Courses*. SIGGRAPH '09. New Orleans, Louisiana: Association for Computing Machinery, 2009.

[11] Joey de Vries. *Shadow-Mapping*. `https://learnopengl.com/Advanced-Lighting/Shadows/Shadow-Mapping`. Accessed: 2020-09-30. 2015.

[12] Lance Williams. “Casting Curved Shadows on Curved Surfaces”. *SIGGRAPH Comput. Graph.* 12.3 (1978.), 270–274.

[13] Fan Zhang i dr. “Parallel-Split Shadow Maps for Large-Scale Virtual Environments”. *Proceedings of the 2006 ACM International Conference on Virtual Reality Continuum and Its Applications*. VRCIA ’06. Hong Kong, China: Association for Computing Machinery, 2006., 311–318.

Dodatak A
Isječci kodova

```
using Entity = std::uint32_t;
```

Kod A.1: Definicija entiteta

```
void CullingSystem::Cull(const Camera& camera)
{
    DirectX::BoundingFrustum frustum = camera.Frustum();

    for (auto& e : entities)
    {
        const AABB& bbox = ecs->GetComponent<AABB>(e);
        bool prevVisible = ecs->HasComponent<Visible>(e);
        bool intersectsFrustum = frustum.Intersects(bbox.bb);
        if (intersectsFrustum && !prevVisible)
                ecs->AddComponent<Visible>(e);
        else if (!intersectsFrustum && prevVisible)
                ecs->RemoveComponent<Visible>(e);
    }
}
```

Kod A.2: Primjer jednostavnog sustava u ECS arhitekturi

```
Entity light = ecs.CreateEntity();
ecs.AddComponent<DirectionalLight>(light);
```

Kod A.3: Stvaranje objekta u ECS-u

```
float CalcShadowFactor_PCF3x3(SamplerComparisonState samShadow,
    Texture2D shadowMap,
    float4 shadowPosH, int smSize, float softness)
```

```
{
    shadowPosH.xyz /= shadowPosH.w;

    float depth = shadowPosH.z;

    const float dx = 1.0f / smSize;

    float percentLit = 0.0f;

    float2 offsets[9] =
    {
        float2(−dx,−dx),float2(0.0f,−dx),float2(dx,−dx),
        float2(−dx,0.0f),float2(0.0f,0.0f),float2(dx,0.0f),
        float2(−dx,+dx),float2(0.0f,+dx),float2(dx,+dx)
    };

    [unroll]
    for (int i = 0; i < 9; ++i)
    {
      offsets[i] = offsets[i] ∗ float2(softness,softness);
        percentLit += shadowMap.SampleCmpLevelZero(samShadow,
            shadowPosH.xy + offsets[i], depth).r;
    }
    return percentLit /= 9.0f;
}
```

Kod A.4: PCF 3x3 filtriranje

```
struct VS_OUTPUT
{
    float4 Pos : SV_POSITION;
};
float4 main(VS_OUTPUT pin) : SV_TARGET
{
    float depth = pin.Pos.z / pin.Pos.w;
    return float4(depth, depth∗depth,0,0);
}
```

Kod A.5: Sjenčar fragmenta za konstrukciju mapu sjene u VSM-u

```
float Linstep(float a, float b, float v)
{
    return saturate((v − a) / (b − a));
}
float ReduceLightBleeding(float pmax, float amount)
{
    // Remove the [0, amount] tail and linearly rescale (amount, 1].
    return Linstep(amount, 1.0f, pmax);
}
float Chebyshev(float2 moments, float depth)
{
    if (depth <= moments.x)
            return 1.0;
    float variance = moments.y − (moments.x ∗ moments.x);
    float d = depth − moments.x; // attenuation
    float pMax = variance / (variance + d ∗ d);
    pMax = max(pMax, 0.01);
    return pMax;
}
float CalcShadowFactorVSM(SamplerState linearSampler, Texture2D shadowMap,
    float4 shadowPosH, float lightBleedingReduction)
{
    shadowPosH.xyz /= shadowPosH.w;

    float fragDepth = shadowPosH.z;
    float lit = 0.0f;
    float2 moments = shadowMap.Sample(linearSampler, shadowPosH.xy).rg;

    float p = Chebyshev(moments, fragDepth);
    p = ReduceLightBleeding(p, lightBleedingReduction);
    lit = max(p, fragDepth <= moments.x);

    return lit;
}
```

Kod A.6: Izračun vidljivosti za VSM algoritam

```
float CalcShadowFactorESM(SamplerState linearSampler, Texture2D shadowMap,
    float4 shadowPosH, float exponent)
{
    shadowPosH.xyz /= shadowPosH.w;
    float fragDepth = shadowPosH.z;
   fragDepth = 2 * fragDepth - 1;
    float moment = shadowMap.Sample(linearSampler, shadowPosH.xy).r;  //exp(depth);
   float visibility = exp(-exponent * fragDepth) * moment;
   return clamp(visibility, 0, 1);
}
```

Kod A.7: Izračun vidljivosti za ESM algoritam

```
float CalcShadowFactorEVSM(SamplerState linearSampler, Texture2D shadowMap,
    float4 shadowPosH, float lightBleedReduction, float exponent)
{
    shadowPosH.xyz /= shadowPosH.w;
    float fragDepth = shadowPosH.z;

    float shadow = 0.0;
    float4 moments = shadowMap.Sample(linearSampler, shadowPosH.xy); // pos, pos^2, neg,
      neg^2

   fragDepth = 2 * fragDepth - 1;
    float pos = exp(exponent * fragDepth);
    float neg = -exp(-exponent * fragDepth);

    float posShadow = Chebyshev(moments.xy, pos);
    float negShadow = Chebyshev(moments.zw, neg);

    posShadow = ReduceLightBleeding(posShadow, lightBleedReduction);
    negShadow = ReduceLightBleeding(negShadow, lightBleedReduction);

    shadow = min(posShadow, negShadow);
    return shadow;
}
```

Kod A.8: Izračun vidljivosti za EVSM algoritam

```
#define BLOCKER_SEARCH_SAMPLES 16

void BlockerSearch(out float avgBlockerDepth,
    out float numBlockers,
    Texture2D tDepthMap,
    SamplerState pointSampler,
    float2 uv,
    float z,
    float2 searchRegionRadius)
{
    float blockerSum = 0;
    numBlockers = 0;

    for (int i = 0; i < BLOCKER_SEARCH_SAMPLES; ++i)
    {
        float2 offset = PoissonSamples16[i] * searchRegionRadius;
        float shadowMapDepth = tDepthMap.SampleLevel(pointSampler, uv + offset, 0);

        if (shadowMapDepth < z - 0.001)
        {
            blockerSum += shadowMapDepth;
            numBlockers++;
        }
    }
    avgBlockerDepth = blockerSum / numBlockers;
}
```

Kod A.9: Računanje zaklonjenosti u PCSS algoritmu

```
class ShadowMapBase : public SystemBase
{
public:
    ShadowMapBase();
    ShadowMapBase(const ShadowMapBase&) = delete;
    ShadowMapBase& operator=(const ShadowMapBase&) = delete;
    ~ShadowMapBase() = default;
    void CalculateSceneBoundingSphere();
    void BindConstants(ID3D11DeviceContext* context) const;
    void Update(ID3D11DeviceContext* context, const DirectionalLight& light, const Camera& camera);
    virtual void ImguiQueries(ID3D11Device* device) = 0;
    virtual void RenderDepthMap(ID3D11DeviceContext* context) = 0;
    void RenderDebugQuad(ID3D11DeviceContext* context);

protected:
    void CommonImgui(ID3D11Device* device);
    void RenderDepthMapNonTransparent(ID3D11DeviceContext* context);
    void RenderDepthMapTransparent(ID3D11DeviceContext* context);

private:
    //lightview and lightprojection
    std::pair<DirectX::XMMATRIX, DirectX::XMMATRIX> SceneLightViewProjection(const DirectionalLight& light);
    std::pair<DirectX::XMMATRIX, DirectX::XMMATRIX> CameraLightViewProjection(const DirectionalLight& light, const Camera& camera);
    virtual void UpdateParametersCBuffer(ID3D11DeviceContext* context);

protected:

    std::unique_ptr<Texture2D> depthMap;
    D3D11_VIEWPORT mViewport;

    DirectX::BoundingSphere boundingSphere;
    DirectX::BoundingBox viewFrustumBoundingBox;

    //states
    std::unique_ptr<SamplerState> shadowSampler;
    std::unique_ptr<SamplerState> pointSampler;
    std::unique_ptr<RasterizerState> rasterizerState;

    //shaders
    std::pair<VertexShader, PixelShader> shadowShader;
    InputLayout shadowLayout;
    std::pair<VertexShader, PixelShader> transparentShadowShader;
    InputLayout transparentShadowLayout;
    std::pair<VertexShader, PixelShader> debugQuadShader;

    //for pcss for now
    FLOAT frustumWidth, frustumHeight, lightZNear, lightZFar;

    //imgui
    INT currentAlgorithm;
    UINT depthMapSize;
    INT lastDepthBias;
    FLOAT lastSlopeScaleDepthBias;
    FLOAT softness;
    bool transparent;
    bool debugQuad;
    bool frustumVPMatrix;
};
```

Kod A.10: Bazna apsktraktna klasa za sve sustave mape sjena. Ovo se ne odnosi na kaskadne mape sjena koje imaju svoju zasebnu klasu.

```
class CascadedShadowMapsSystem : public SystemBase
{
public:

    static constexpr UINT MAX_CASCADES_NUM = 3u;

    explicit CascadedShadowMapsSystem(ID3D11Device* device);

    void BindConstants(ID3D11DeviceContext* context) const;

    void Update(ID3D11DeviceContext* context, const DirectionalLight& light, const Camera& camera);

    void RenderDepthMaps(ID3D11DeviceContext* context);

    void RenderDebugQuad(ID3D11DeviceContext* context);

    void ImguiQueries(ID3D11Device* device);

private:

    std::array<DirectX::XMFLOAT4X4, MAX_CASCADES_NUM> RecalculateProjectionMatrices(const Camera& camera);

    std::array<DirectX::XMMATRIX, MAX_CASCADES_NUM> CalculateLightViewProjectionMatrices(const DirectionalLight& light, const Camera& camera);

    std::array<DirectX::XMMATRIX, MAX_CASCADES_NUM> CalculateStableLightViewProjectionMatrices(const DirectionalLight& light, const Camera& camera);

    void InitializeZBuffer(ID3D11Device* device, UINT size);

    void InitializeDepthMaps(ID3D11Device* device, UINT size);

    void RenderNonTransparentDepthMaps(ID3D11DeviceContext* context);

    void RenderTransparentDepthMaps(ID3D11DeviceContext* context);

private:

    D3D11_VIEWPORT mViewport;
    std::unique_ptr<Texture2D> depthMaps;
    std::unique_ptr<Texture2D> zBuffer;
    std::array<Microsoft::WRL::ComPtr<ID3D11RenderTargetView>, MAX_CASCADES_NUM> rtvs;
    std::array<DirectX::XMMATRIX, MAX_CASCADES_NUM> projs;
    std::array<FLOAT, MAX_CASCADES_NUM> splitDistances;
    //states
    std::unique_ptr<SamplerState> shadowSampler;
    std::unique_ptr<RasterizerState> rasterizerState;

    //shaders
    std::pair<VertexShader, PixelShader> shadowShader;
    InputLayout shadowLayout;
    std::pair<VertexShader, PixelShader> transparentShadowShader;
    InputLayout transparentShadowLayout;
    std::pair<VertexShader, PixelShader> debugQuadShader;

    //imgui stuff
    INT currentAlgorithm;
    UINT depthMapSize;
    INT lastDepthBias;
    FLOAT lastSlopeScaleDepthBias;
    bool transparent;
    bool debugQuad;
    bool visualizeCascades;
    FLOAT splitLambda;
    bool stabilize;
    bool blend;
    bool mapBasedSelection;
    FLOAT nearFactor;
    FLOAT farFactor;
    FLOAT lightDistanceFactor;
};
```

Kod A.11: Klasa za kaskadne mape sjena.

```
std::array<DirectX::XMFLOAT4X4, CascadedShadowMapsSystem::MAX_CASCADES_NUM> CascadedShadowMapsSystem::RecalculateProjectionMatrices(const
        Camera& camera)
{
    float nearPlane = camera.Near();
    float farPlane = camera.Far();
    auto [fov, ar] = camera.GetFovAndAspectRatio();

    float f = 1.0f / MAX_CASCADES_NUM;
    for (UINT i = 0; i < splitDistances.size(); i++)
    {
        float fi = (i + 1) * f;
        float l = nearPlane * pow(farPlane / nearPlane, fi);
        float u = nearPlane + (farPlane - nearPlane) * fi;
        splitDistances[i] = l * splitLambda + u * (1.0f - splitLambda);
    }

    std::array<DirectX::XMFLOAT4X4, MAX_CASCADES_NUM> projectionMatrices;
    DirectX::XMStoreFloat4x4(&projectionMatrices[0], DirectX::XMMatrixPerspectiveFovLH(fov, ar, nearPlane, splitDistances[0]));
    for (UINT i = 1; i < projectionMatrices.size(); ++i)
        DirectX::XMStoreFloat4x4(&projectionMatrices[i], DirectX::XMMatrixPerspectiveFovLH(fov, ar, splitDistances[i-1], splitDistances[i]));

    return projectionMatrices;
}
```

Kod A.12: Računanje matrica projekcije za svaku kaskadu iz kojih se kasnije izvuku vrhovi krnje piramide

```
std::array<DirectX::XMMATRIX, CascadedShadowMapsSystem::MAX_CASCADES_NUM> CascadedShadowMapsSystem::CalculateLightViewProjectionMatrices(
        const DirectionalLight& light, const Camera& camera)
{
    std::array<DirectX::XMFLOAT4X4, MAX_CASCADES_NUM> projectionMatrices = RecalculateProjectionMatrices(camera);
    std::array<DirectX::XMMATRIX, MAX_CASCADES_NUM> lightViewProjectionMatrices{};

    for (UINT i = 0; i < MAX_CASCADES_NUM; ++i)
    {
        ///frustum in world space
        BoundingFrustum frustum(DirectX::XMLoadFloat4x4(&projectionMatrices[i]));
        frustum.Transform(frustum, DirectX::XMMatrixInverse(nullptr, camera.View()));

        ///get frustum corners
        std::array<DirectX::XMFLOAT3, frustum.CORNER_COUNT> corners;
        frustum.GetCorners(corners.data());
        DirectX::XMVECTOR frustumCenter = DirectX::XMVectorSet(0, 0, 0, 0);
        for (UINT i = 0; i < corners.size(); ++i)
        {
            frustumCenter = frustumCenter + DirectX::XMLoadFloat3(&corners[i]);
        }
        frustumCenter /= static_cast<float>(corners.size());

        float radius = 0.0f;

        for (UINT i = 0; i < corners.size(); ++i)
        {
            float dist = DirectX::XMVectorGetX(XMVector3Length(DirectX::XMLoadFloat3(&corners[i]) - frustumCenter));
            radius = std::max(radius, dist);
        }
        radius = std::ceil(radius * 8.0f) / 8.0f;

        ///calculating light view matrix
        XMVECTOR lightDir = DirectX::XMVector3Normalize(XMLoadFloat3(&light.direction));
        //static const XMVECTOR zero = DirectX::XMVectorSet(0, 0, 0, 1);
        XMVECTOR up = camera.Right(); // XMVectorSet(0.0f, 1.0f, 0.0f, 0.0f);
        XMVECTOR lightPos = -lightDistanceFactor * radius * lightDir + frustumCenter;

        ///NEW view matrica
        XMMATRIX V = DirectX::XMMatrixLookAtLH(lightPos, frustumCenter, up);

        ///transform frustum corners in light view space to calculate AABB for orthographic projection
        FLOAT l = std::numeric_limits<float>::max();
        FLOAT b = std::numeric_limits<float>::max();
        FLOAT n = std::numeric_limits<float>::max();
```

```
        FLOAT r = std::numeric_limits<float>::min();
        FLOAT t = std::numeric_limits<float>::min();
        FLOAT f = std::numeric_limits<float>::min();

        for (auto& corner : corners)
        {
            //transform corners into light space
            XMStoreFloat3(&corner, DirectX::XMVector3TransformCoord(DirectX::XMLoadFloat3(&corner), V));
            l = std::min(l, corner.x);
            r = std::max(r, corner.x);

            b = std::min(b, corner.y);
            t = std::max(t, corner.y);

            n = std::min(n, corner.z);
            f = std::max(f, corner.z);
        }

        XMMATRIX P = DirectX::XMMatrixOrthographicOffCenterLH(l, r, b, t, nearFactor*n, farFactor*f);

        XMMATRIX lightviewprojection = V * P;
        lightViewProjectionMatrices[i] = lightviewprojection;
    }
    return lightViewProjectionMatrices;
}
```

Kod A.13: Računanje matrica pogleda i projekcije svjetla za svaku kaskadu

```
float viewDepth = pin.ViewSpacePos.z;
    //without cascade blending
    if (blend == 0)
    {
        for (uint i = 0; i < MAX_CASCADES_NUM; ++i)
        {
            float4 ShadowPos = mul(float4(pin.PositionWS, 1.0f), lightviewprojection[i]);
            ShadowPos.xy = 0.5 * ShadowPos.xy + 0.5;
            ShadowPos.y = 1 - ShadowPos.y;
            ShadowPos.xyz /= ShadowPos.w;

            if (mapBasedSelection)
            {
                if (ShadowPos.x >= 0 && ShadowPos.x <= 1 && ShadowPos.y >= 0 && ShadowPos.y <= 1)
                {
                    shadowFactor = CSMCalcShadowFactorPCF(shadowSampler, depthMap, i, ShadowPos, pcfType, shadowMapSize);
                    if (visualize)
                    {
                        float4 color = float4(0, 0, 0, 1);
                        color[i] = 1;
                        return color * shadowFactor;
                    }
                    break;
                }
            }
            else
            {
                if (viewDepth < splits[i])
                {
                    shadowFactor = CSMCalcShadowFactorPCF(shadowSampler, depthMap, i, ShadowPos, pcfType, shadowMapSize);
                    if (visualize)
                    {
                        float4 color = float4(0, 0, 0, 1);
                        color[i] = 1;
                        return color * shadowFactor;
                    }
                    break;
                }
            }
        }
        D = D * shadowFactor;
        S = S * shadowFactor;
        return texColor * float4(A + D, 1.0) + float4(S, 1.0);
    }
```

```
  //blending
  else
  {
    for (uint i = 0; i < MAX_CASCADES_NUM; ++i)
    {
      if (viewDepth < splits[i] * 0.9 || i == MAX_CASCADES_NUM - 1)
      {

        float4 ShadowPos = mul(float4(pin.PositionWS, 1.0f), lightviewprojection[i]);
        ShadowPos.xy = 0.5 * ShadowPos.xy + 0.5;
        ShadowPos.y = 1 - ShadowPos.y;
        ShadowPos.xyz /= ShadowPos.w;

        shadowFactor = CSMCalcShadowFactorPCF(shadowSampler, depthMap, i, ShadowPos, pcfType, shadowMapSize);
        if (visualize)
        {
          float4 color = float4(0, 0, 0, 1);
          color[i] = 1;
          return color * shadowFactor;
        }

        D = D * shadowFactor;
        S = S * shadowFactor;
        return texColor * float4(A + D, 1.0) + float4(S, 1.0);
      }


      else if (viewDepth < splits[i] * 1.1 && i != MAX_CASCADES_NUM - 1)
      {
        float4 ShadowPos = mul(float4(pin.PositionWS, 1.0f), lightviewprojection[i]);
        ShadowPos.xy = 0.5 * ShadowPos.xy + 0.5;
        ShadowPos.y = 1 - ShadowPos.y;
        ShadowPos.xyz /= ShadowPos.w;

        shadowFactor = CSMCalcShadowFactorPCF(shadowSampler, depthMap, i, ShadowPos, pcfType, shadowMapSize);

        float4 ShadowPos2 = mul(float4(pin.PositionWS, 1.0f), lightviewprojection[i + 1]);
        ShadowPos2.xy = 0.5 * ShadowPos2.xy + 0.5;
        ShadowPos2.y = 1 - ShadowPos2.y;
        ShadowPos2.xyz /= ShadowPos2.w;

        float shadowFactor2 = CSMCalcShadowFactorPCF(shadowSampler, depthMap, i + 1, ShadowPos2, pcfType, shadowMapSize);

        if (visualize)
        {
          float4 color = float4(0, 0, 0, 1);
          float4 color2 = float4(0, 0, 0, 1);
          color[i] = 1;
          color2[i + 1] = 1;
          return lerp(color * shadowFactor, color2 * shadowFactor2, 1 - (1.1 - viewDepth / splits[i]) / 0.2);
        }
        shadowFactor = lerp(shadowFactor,shadowFactor2, 1 - (1.1 - viewDepth / splits[i]) / 0.2);
        D = D * shadowFactor;
        S = S * shadowFactor;

        return texColor * float4(A + D, 1.0) + float4(S, 1.0);
      }
    }
  }
  return float4(1, 1, 1, 1);
}
```

Kod A.14: Računanje vidljivosti kod kaskadnih mapa sjena

Prikaz sjena korištenjem mape sjene

Sažetak

U ovom radu su predstavljene neke od poboljšanih varijanti algoritma mape sjene. Obrađeni algoritmi se mogu podijeliti na filtrirane mape sjena i kaskadne mape sjena koji su predstavnik skupine particijskih algoritama. U skupini filtriranih mapa sjena se razlikuju algoritmi bez predfiltriranja s fiksnom i s varijabilnom veličinom filtra te algoritmi s predfiltriranjem. Opisana je teorijska pozadina te nedostatci i moguća rješenja svakog algoritma. Napravljena je implementacija koja omogućuje usporedbu svih obrađenih algoritama i analizu utjecaja raznih parametara na kvalitetu sjene.

Ključne riječi: sjene, mapa sjene, kaskadne mape sjena, filtrirane mape sjena, meke sjene, SSM, PCF, VSM, ESM, EVSM, PCSS, CSM, PSSM, C++, DirectX

Shadows with Shadow Mapping

Abstract

In this paper, some of the improved variants of the shadow map algorithm are presented. Those algorithms can be divided into filtered shadow maps and cascaded shadow maps which is representative of z-partition algorithms. In the group of filtered shadow maps, algorithms without prefiltering with a fixed and with a variable filter size and algorithms with prefiltering are distinguished. The theoretical background is described, as well as the shortcomings and possible solutions of each algorithm. An implementation has been made that allows comparison of all processed algorithms and the analysis of the influence of various parameters on the shadow quality.

Keywords: shadows, shadow maps, cascaded shadow maps, filtered shadow maps, soft shadows, SSM, PCF, VSM, ESM, EVSM, PCSS, CSM, PSSM, C++, DirectX