Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Matevž Ropret

Realnočasovna simulacija indirektne osvetlitve

diplomsko delo

univerzitetni študijski program prve stopnje
računalništvo in informatika

Mentor: doc. dr. Matija Marolt

Ljubljana 2013

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Univerza *v Ljubljani*
Fakulteta *za računalništvo in informatiko*

Tržaška 25
1001 Ljubljana, Slovenija
telefon: 01 476 84 11
01 476 83 87
faks: 01 426 46 47
01 476 87 11
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 00120/2013
Datum: 11.04.2013

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **MATEVŽ ROPRET**

Naslov: **REALNOČASOVNA SIMULACIJA INDIREKTNE OSVETLITVE**
SIMULATION OF INDIRECT ILLUMINATION IN REAL-TIME

Vrsta naloge: Diplomsko delo univerzitetnega študija prve stopnje

Tematika naloge:

V okviru diplomske naloge preučite področje realnočasovne simulacije indirektne osvetlitve. Na grafični procesni enoti implementirajte postopek, v katerem si pri izračunu osvetlitve pomagate s prostorsko podatkovno strukturo, in ocenite učinkovitost posameznih faz algoritma.

Mentor:

doc. dr. Matija Marolt

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

Dekan:

prof. dr. Nikolaj Zimic

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Matevž Ropret, z vpisno številko **63100231**, sem avtor diplomskega dela z naslovom:

Realnočasovna simulacija indirektne osvetlitve

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom doc. dr. Matije Marolta,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki "Dela FRI".

V Ljubljani, dne 11. januarja 2011 Podpis avtorja:

Kazalo

Povzetek

Abstract

1 Uvod **1**

1.1 Realnočasovna simulacija grafike 2
1.2 Ne-realnočasovna simulacija grafike 2
1.3 Kaj sploh želimo doseči? 3
1.4 Cilj naloge 3
1.5 Struktura naloge 3
1.6 Zgodovina simulacije svetlobe v realnem času 4
1.6.1 Obdobje pred polnimi poligoni (1980) 4
1.6.2 Obdobje fiksnega cevovoda (1987) 4
1.6.3 Obdobje programabilnega cevovoda (2000) 5
1.6.4 Dosedanji razvoj realnočasovne indirektne osvetlitve . . 6
1.6.5 Razlogi za počasen razvoj realnočasovne indirektne osvetlitve 8

2 Implementacija dinamične indirektne osvetlitve **11**

2.1 Uvod 11
2.2 Prva faza: generiranje senc 11
2.3 Druga faza: Zapis scene v medpomnilnik in injektiranje svetlobe 13
2.4 Tretja faza: izris scene 16
2.5 Četrta faza: vokselizacija 16

2.6 Peta faza: simulacija odboja svetlobe 19
2.6.1 Priprava za algoritem 19
2.6.2 Streljanje žarkov 19
2.7 Šesta faza: zlitje direktne in indirektne osvetlitve 24
2.8 Meritve porabe časa 25
2.9 Podporna tehnologija za realizacijo 26
2.9.1 Visokonivojski pregled podporne tehnologije 26
2.9.2 Skupni modul 27
2.9.3 Grafični moduli 27
2.9.4 Modul za nalaganje 3D modelov 28
2.9.5 Modul za pomoč pri uporabi operacijskega sistema . . 29
2.9.6 Izvajalni modul 29
2.9.7 Koordinatni sistemi 29
2.10 Primeri 31

3 Zaključek **33**

Seznam kratic

Kratica	Angleško	Prevod
CPE	/	centralna procesna enota
FPS	frame per second	število slik na sekundo
GPE	/	grafična procesna enota
GS	geometry shader	senčilnik geometrije
IB	index buffer	indeksni medpomnilnik
IL	input layout	zgradba vhoda v VS
PS	pixel shader	senčilnik pikslov
RT	render target	izrisovalna površina
VB	vertex buffer	ogljiščni medpomnilnik
VS	vertex shader	senčilnik ogljišč
ZB	Z-Buffer	globinska tekstura

Povzetek

Do nedavnega se je v računalniški grafiki v realnem času simulirala samo direktna osvetlitev, torej svetloba, ki pride direktno iz luči do površine. Seveda se v realnem svetu svetloba od površin odbije in ravno to je nujno potrebno, če želimo izrisati virtualne scene, ki so podobne realnemu svetu. Kot bomo videli, je simulacijo odboja svetlobe zelo težko implementirati učinkovito. V tem delu najprej predstavim dosedanje realnočasovne metode za izris indirektne osvetlitve, nato pa razložim svojo metodo. Bistven del moje metode je to, da ni potrebno ničesar izračunati vnaprej. To omogoča indirektno osvetlitev na povsem dinamičnih scenah. Moja metoda v trenutni fazi še ni povsem praktična za uporabo v igrah, a bi se jo zagotovo dalo še zelo izboljšati.

Abstract

Until recently only the light which comes directly from a light source to a surface could have been simulated in realtime. Of course in the real world, light bounces off from surfaces and creates indirect illumination which is of the key importance to render a virtual scene which looks realistic. We will see that indirect illumination is a difficult algorithm to implement efficiently. In this work I have presented some existing real-time methods for computing indirect illumination and then I have explained my algorithm for indirect illumination simulation. The key thing about my algorithm is that it supports fully dynamic scenes which means that we don't have to bake anything in advance. In the current phase my method hasn't been ready yet to be implemented in a video game, but with further research it could be vastly improved.

Poglavje 1

Uvod

Ljudje želimo izboljšati obstoječe stvari in ravno v računalništvu so novosti nekaj povsem vsakdanjega. Zahvaljujoč se ekstremno hitremu povečanju zmogljivosti vseh vrst računalnikov lahko dandanes vsakdo doma, če ima le voljo, ustvari izjemno kompleksne programe. Velikokrat nas omejuje zgolj naše znanje. A ni vedno tako. Če vstopimo na področje fizikalno pravilne simulacije sveta, pa hitro ugotovimo, da tudi današnji računalniki le niso tako zmogljivi. Velikokrat je stvari potrebno narediti zgolj perceptualno realno, saj nimamo na voljo dovolj močnih računalnikov.

Področje računalniške grafike brez dvomno spada v kategorijo, kjer je pomemben zgolj perceptualni rezultat, ki ga ljudje zaznamo z očmi. Ali je določeni piksel na zaslonu svetlejši ali temnejši za en odtenek, ljudje mnogokrat nismo sposobni zaznati, zato lahko na tem področju uporabljamo približke, ki so na drugih področjih (na primer v bančništvu) nesprejemljivi.

Če se sprehodimo skozi celotno področje računalniške grafike lahko hitro vidimo, da se deli na dve ogromni podpodročji: **realnočasovno** in **nerealnočasovno**.

1.1 Realnočasovna simulacija grafike

Fraza **realen čas** se uporablja za programe, ki se instantno odzovejo na uporabnikovo zahtevo. V to skupino spadajo npr. urejevalniki besedil, a za naše potrebe so bolj zanimive videoigre in razne medicinske vizualizacije. Videoigre morajo za prijetno izkušnjo sliko izrisati vsaj 30-krat na sekundo oziroma bolje je 60-krat. To pomeni, da imamo za izris ene celotne vidne scene v navideznem svetu na voljo zelo malo časa (če ciljamo 60 FPS, to pomeni da, scene ne smemo izrisovati več kot 16.6ms).

Ravno zaradi te ekstremne časovne omejitve se v igrah ne uporablja fizikalno pravilna simulacija svetlobe (sledenje nekaj milijardam fotonov je sicer preprost problem za simulacijo, a je zelo časovno zahteven). Ravno zaradi tega se v igrah dandanes uporablja rasterska grafika. Princip je enostaven: izriši trikotnik, poglej, katera luč ga osvetljuje, izračunaj odziv materiala in zapiši trikotnik v teksturo (seznam pikslov, ponavadi v 2 dimenzijah, s čimer lahko predstavimo sliko), ki se potem prikaže na zaslonu. Z vidika strojne opreme je to odlična poenostavitev, saj so grafični elementi tako popolnoma neodvisni eden od drugega. Ta neodvisnost je bistvena značilnost, okoli katere so zgrajene vse grafične kartice, tako v preteklosti kot tudi dandanes, saj omogoča enostavno masovno paralelizacijo izvajanja programov.

1.2 Ne-realnočasovna simulacija grafike

V to kategorijo spada vsa računalniška grafika, kjer si lahko vzamemo dolgo časa za računanje ene slike (naprimer 1 uro). Najbolj očiten primer za to so filmi. Ker ti niso interaktivna vsebina, si lahko za izris ene same slike privoščimo absurdno dolgo. Dober primer je film Transformers: Dark of the Moon [5], kjer se je za najbolj kompleksno sceno porabilo 122 ur za eno sliko. Tukaj lahko za osvetlitev uporabimo sledenje fotonom skozi sceno, kar odlično simulira delovanje realne svetlobe in nam omogoča izris zelo realnih scen.

1.3 Kaj sploh želimo doseči?

V realnem svetu se svetloba vede zelo kompleksno. Lahko se zgolj odbije od površine (na primer pri prevodnih snoveh) ali pa se del svetlobe odbije, del pa vstopi v objekt in potem spet izstopi ven iz objekta (na primer človeška koža). Eden od najbolj opaznih lastnosti svetlobe je odboj le-te od površin. Če z lučjo posvetimo v steno, se ne bo osvetlil le del stene, ampak tudi okolica, saj se svetloba odbije od stene.

V tem delu se ukvarjam ravno s tem problemom: kako lahko objekte osvetlimo s svetlobo, ki se odbije od objektov v virtualnem svetu. Kot bomo videli, je ta problem zelo težko učinkovito realizirati in velja za enega od svetih gralov realno časovne grafike.

1.4 Cilj naloge

Končni cilj je simulacija enega odboja svetlobe od vseh objektov. S to svetlobo osvetlimo vse objekte na sceni. Bistvenega pomena je to, da so lahko vsi objekti povsem dinamični (se lahko premikajo).

1.5 Struktura naloge

Najprej bomo na hitro pregledali, na kateri dve področji se deli računalniška grafika. Temu sledi zgodovinski pregled razvoja realnočasovne grafike. Nato sledi pregled obstoječih metod za simulacijo indirektne osvetlitve in razloge za odsotnost indirektne osvetlitve v igrah do nedavnega. Sledi opis moje metode za izračun indirektne osvetlitve po fazah. Temu sledijo časovne meritve, končni rezultati in možnosti za izboljšanje. Na koncu je še opis mojega programa in modulov, ki so potrebni za realizacijo algoritma, ki ga predstavljam v tem delu.

1.6 Zgodovina simulacije svetlobe v realnem času

1.6.1 Obdobje pred polnimi poligoni (1980)

Prvi računalniki so bili sposobni zgolj izrisa točk, črt in krivulj. Zapolnjenih površin (npr. trikotnikov) niso bili sposobni izrisovati. Zaradi takega abstraktnega prikaza sveta tu simulacija svetlobe niti ni imela smisla.

1.6.2 Obdobje fiksnega cevovoda (1987)

Sčasoma so grafične kartice postale samostojen del strojne opreme, neodvisen od CPE ali ostalih komponent. Vojna med proizvajalci GPE je na začetku povzročala hudo zmedo tako za programerje kot tudi za kupce iger. Funkcionalnost GPE je bila namreč povsem različna, saj ni bilo nobenega standarda, ki bi se ga proizvajalci držali. Sčasoma sta se oblikovala standarda OpenGL [9] in Direct3D [10], ki sta zahtevala določeno funkcionalnost od vseh GPE. Oba standarda uporabljata tako imenovani **grafični cevovod**.

Grafični cevovod je zaporedje več faz, skozi katere potujejo podatki. V teh fazah se podatki pretvorijo iz primitivnih objektov (npr. trikotnikov) v piksle na zaslonu, ki potem predstavljajo nek objekt v navideznem svetu. Grafični cevovodi najprej niso bili programabilni: lahko se je sicer nastavljalo določene parametre, a vsa funkcionalnost je bila shranjena v strojni opremi. To je omogočalo ekstremne optimizacije s strani proizvajalcev GPE. Slaba stran tega je bila, da so bili programerji hudo omejeni pri ustvarjanju raznih efektov. Če nekega efekta namreč ni bilo implementiranega v strojni opremi, ga ni bilo mogoče na noben način realizirati. Vse tehnike osvetlitve so bile že pripravljene vnaprej, zato programerji niso imeli veliko izbire pri fiksnem cevovodu.

Slika 1.1: Levo Starglider [6] s 3D vektorsko grafiko vesoljskih objektov, desno Quake3 [7] s fiksnim cevovodom

1.6.3 Obdobje programabilnega cevovoda (2000)

Revolucija v realnočasovni grafiki je prišla na dan z novima standardoma OpenGL 2.0 (2004) in Direct3D 8 (2000). Ta dva sta omogočalo programiranje faze VS in PS. Tukaj je bil nabor ukazov še zelo omejen, a OpenGL in Direct3D sta se zelo razvila in dandanes so poljubno programabilne skoraj vse faze cevovoda. To omogoča programerjem eksperimentiranje z raznimi tehnikami osvetlitve, kot so luči poljubne oblike, razne tehnike za izris senc, zapletene simulacije materialov ...

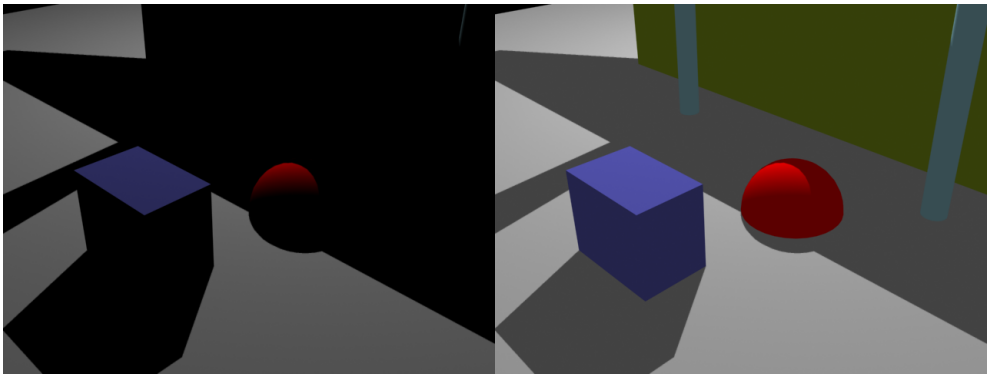
Slika 1.2: Igra Crysis 3 [8] je trenutno vrhunec realistične realnočasovne grafike

1.6.4 Dosedanji razvoj realnočasovne indirektne osvetlitve

Tukaj bomo na hitro pregledali obstoječe metode, ki se delijo na statične in dinamične. Statične metode lahko izgledajo zelo dobro, ampak se lahko uporabljajo samo na objektih, ki se ne morejo premikati (npr. hiše, ceste). Dinamične metode pa seveda podpirajo realno indirektno osvetlitev od vseh objektov, tudi tistih, ki se premikajo.

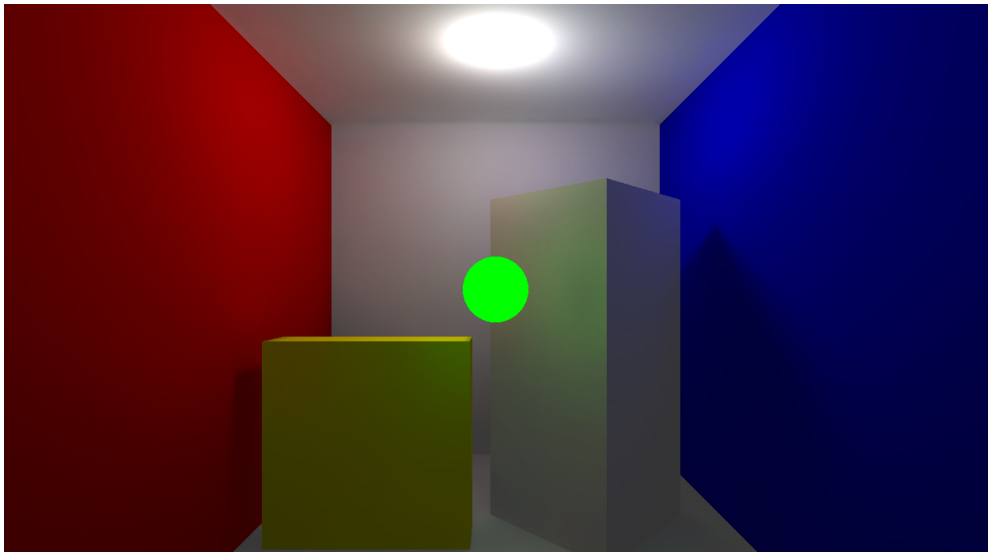
Statične metode

Najbolj preprosta metoda je nedvomno preprosta ambientna svetloba. To vnesemo v sceno tako, da zgolj prištejmo neko vrednost vsakemu pikslu. Ta tehnika je najhitrejša, a seveda daje zelo nerealistične rezultate, saj ne upošteva barve objektov. Veliko bolj realne rezultate dobimo, če uporabimo



Slika 1.3: Levo slika brez ambientne svetlobe, desno pa z njo

bolj realen simulator svetlobe. Tega poženemo, ko imamo postavljene vse statične objekte v sceni in zapečemo svetlobo v teksture. Ta proces opravimo samo enkrat, ko program teče v realnem času, pa samo beremo iz teh tekstur. Največji problem te metode je statičnost: noben objekt, ki je zapečen v teksturo, se ne sme premakniti. Ker je svetloba izračunana vnaprej, se osvetlitev ne bo spremenila, v primeru da se bo objekt premaknil. Ta metoda se dandanes vedno manj uporablja.



Slika 1.4: Vnaprej izračunana osvetlitev daje zelo lepe rezultate, a se ne simulira v realnem času, tako da ni primerna za dinamične scene [11]

Seveda obstaja še več metod, na primer ročno postavljanje ambientnih luči, a nobena od teh metod ne daje zelo realnih rezultatov, še posebej če imamo v sceni velike dinamične objekte.

Dinamične metode

V času pisanja te diplomske ni veliko dobrih rešitev, primernih za realno časovne aplikacije. Izstopajo tri: prva je Crytekov *Light propagation volumes* [4], ki se uporabljajo v igri Crysis 2. Pri tej metodi v volumetrično strukturo shranijo osvetlitev pridobljeno iz površine objektov in jo propagirajo po volumnu.

Druga metoda je *Deferred radiance transfer volumes* [2] podjetja Massive, ki se uporablja v igri Far Cry 3. Za to metodo po celotnem svetu enakomerno razporedijo posebne točke (samo eno vertikalno). Nato vsakič, ko izrišejo sliko, posodobijo samo eno to točko. To storijo tako, da v vseh 6 smereh okoli točke izrišejo sceno in zapišejo osvetlitev v to točko. Ker je indirektna osvetlitev že sama po sebi zelo nerazločna v primerjavi z direktno

osvetlitvijo, lahko tudi s počasnim posodabljanjem posameznih točk dobijo zelo lep približek.

Slika 1.5: Crytekov *Light propagation volumes*

Tretja metoda je *Voxel cone tracing* [3]. Za to metodo moramo celotno virtualno sceno vokselizirati, tj. shraniti v nekakšno volumetrično podatkovno strukturo. Po tej strukturi za vsak vidni piksel streljamo žarke znotraj nekega stožca, ki ima center v trenutnem pikslu. Pri tem gledamo, kdaj bo žarek v stožcu zadel ob kakšen element v volumetrični strukturi. Zadetke seštejemo in jih lahko artistično ali pa matematično pravilno utežimo. Ta tehnika daje odlične rezultate, ampak je zelo počasna. Moja implementacija temelji na tej metodi.

1.6.5 Razlogi za počasen razvoj realnočasovne indirektne osvetlitve

Od preprostih pobarvanih trikotnikov smo dandanes prišli že zelo daleč, a hitra simulacija vsaj približno realne indirektne osvetlitve je še vedno izven dosega. Razloga za to sta v splošnem dva: smer razvoja GPE skozi zgodovino in počasen razvoj na področju hitrosti pomnilnikov.

Če se osredotočimo na razvoj GPE skozi zgodovino, vidimo, da je arhi-

tektura v splošnem ostala enaka: v srcu GPE je še vedno grafični cevovod. Faze cevovoda so med seboj popolnoma neodvisne, saj prav to omogoča GPE vzporedno računanje velikega števila elementov. Zaradi tega masovnega paralelizma so se GPE razvile tako, da pričakujejo za vsak element v cevovodu enako pot izvajanje programa.

Prav tako je izris enega trikotnika neodvisen od drugega (če bi šli v podrobnosti, bi lahko rekli, da to ni vedno povsem res). Ta neodvisnost velja tudi med celotnimi objekti, ki jih izrisujemo. Vzemimo za primer cesto in avto. Avto ne ve, da obstaja cesta in cesta ne ve, da obstaja avto. In ravno ta neodvisnost predstavlja ogromno težavo za računanje indirektne osvetlitve: če hočemo videti odbito svetlobo od ceste na avtomobilu, mora avto očitno vedeti, da obstaja cesta. To podere celotno načelo neodvisnih objektov. Ne moremo izrisati indirektne osvetlitve na avtomobilu, če prej ne vemo, da obstaja cesta in obratno. Torej moramo celotno sceno (avto in cesto v tem primeru) shraniti v nekakšno podatkovno strukturo. Seveda nam GPE to omogočajo, a to zelo odstopa od učinkovite izrabe GPE. Zaradi tega so mnogi bolj realistični algoritmi za realnočasovno indirektno osvetlitev zelo počasni.

Drugi razlog za počasen razvoj realno časovne indirektne osvetlitve je počasen razvoj hitrosti dostopa do pomnilnika v primerjavi z zmogljivostjo računskih enot. Dandanes veliko hitreje izračunamo kompleksne transcendentne funkcije sproti, kot pa da bi jih izračunali vnaprej in brali iz pomnilnika. Seveda nam je tu v veliko pomoč učinkovita izraba predpomnilnika, a velikost tega je zelo majhna. Če moramo v nekakšno podatkovno strukturo zapisati celotno, ali vsaj del, virtualne scene, hitro vidimo, da nam predpomnilnika zmanjka. Zaradi tega moramo dostopati do glavnega pomnilnika GPE, ki je pa dejanski glavni krivec za počasno delovanje vseh bolj realnih tehnik hitre indirektne osvetlitve. Če bi bil dostop do pomnilnika dandanes instanten, bi brez težav simulirali že zelo realistično indirektno osvetlitev v realnem času.

Poglavje 2

Implementacija dinamične indirektne osvetlitve

2.1 Uvod

Že na začetku sem bil trdno odločen, da hočem imeti realistično indirektno osvetlitev in ne enostavnih približkov. Po pregledu obstoječih tehnik sem se odločil, da bom svoje delo zasnoval podobno kot *Voxel Cone Tracing*. Razlog za to je v zmožnosti realnega odboja svetlobe, saj metoda deluje podobno kot svetloba v resničnem svetu. Bistvo moje metode je v shranjevanju celotne scene v volumetrično teksturo, po kateri se strelja žarke in ugotavlja zadetke, ki se potem uporabijo kot približek za indirektno osvetlitev.

2.2 Prva faza: generiranje senc

Bistveni del realne osvetlitve so sence. Skozi zgodovino so se uporabljale nekako tri skupine tehnik za izrisovanje senc:

- preprosta vnaprej zgenerirana slika (na primer krog, ki je črn v sredini in bel pri robovih), ki se potem projecira na svet;

- geometrijske sence: 3D modelom na sceni poiščemo zunanje robove glede na luč in jih podaljšamo v smeri svetenja luči, nato to podaljšano geometrijo

uporabimo kot masko, da vemo kje so sence;

- mapiranje senc (*shadow mapping*): dandanes prevladuje ta tehnika, saj je najbolj fleksibilna. Za to tehniko moramo sceno izrisati še enkrat iz vidnega polja luči, kot rezultat pa shranimo globino za vsak piksel v neko teksturo. V grafičnem cevovodu nastavimo fazo VS, fazo PS pa izklopimo, nastavimo edino ZB, zato da se vanj zapiše globina. Potem pri risanju scene projeciramo to globinsko teksturo na sceno in primerjamo projecirano globino s teksture z dejansko oddaljenostjo do luči za tisti piksel. Če je globina večja, vemo, da smo v senci. V delu uporabljam to metodo. Sceno najprej izrišem in globino za vsak piksel shranim v teksturo, veliko 1024x1024 pikslov.

Uporabljam tudi preproste mehke sence. Te sem realiziral tako, da pri izrisovanju dejanske scene za vsak piksel izračunam 64 naključnih 2D vektorjev. Te naključne vektorje potem uporabim tako, da pri branju globinske teksture vsakič koordinate branja zamaknem za ta naključni vektor. Seveda je 64 vzorcev za vsak piksel veliko, ampak to ni bistvenega pomena, saj nima vpliva na dejanski algoritem za simulacijo indirektne osvetlitve.

Spodnja enačba prikazuje izračun koordinat teksture, iz katere preberemo globino piksla. $\vec{W}$ predstavlja pozicijo v koordinatah sveta za površino na trenutnem pikslu, LVP predstavlja zmnoženi matriki za kamero in projekcijo luči. Definirana sta, kot prikazuje enačba 2.1. Najprej pretvorimo vektor $\vec{W}$ v homogeni rezalni koordinatni (*homogeneous clip*) sistem z 2.2. Nato opravimo dejansko projekcijo z enačbo 2.3. Na koncu z enačbo 2.4 pretvorimo definicijsko območje $\vec{T}$ iz [-1,1] v [0,1]. Vektor $\vec{T}.xy$ uporabimo za branje globinske teksture, ki smo jo izrisali iz perspektive luči.

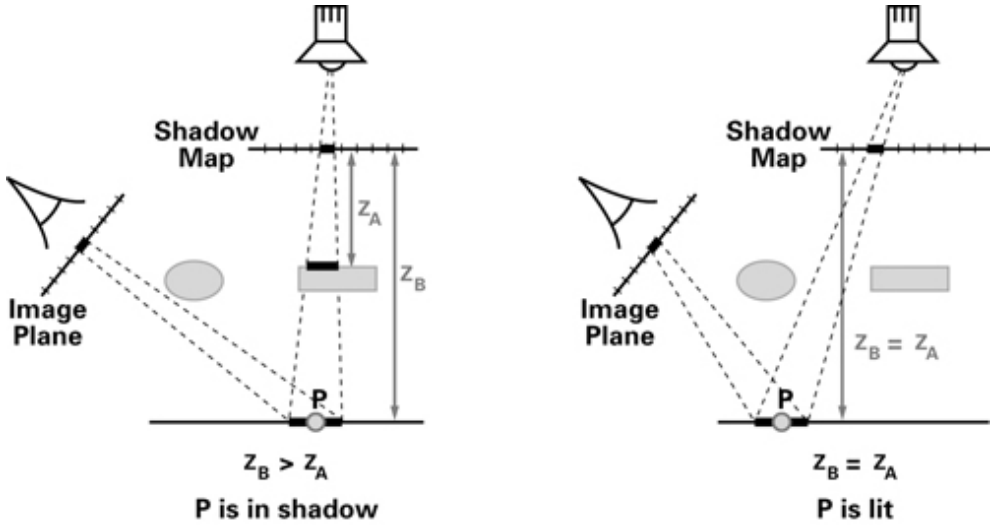
$$\vec{W} = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}, LVP = \begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{2,4} \\ a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} \end{pmatrix} \tag{2.1}$$

$$\vec{T} = \vec{W} * LVP \tag{2.2}$$

$$\vec{T}.xyz = \frac{\vec{T}.xyz}{\vec{T}.w} \tag{2.3}$$

$$\vec{T}.xy = \vec{T}.xy * 0.5 + \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} \tag{2.4}$$

Na koncu vektorju $\vec{T}.xy$ dodam še naključni odmik vsako iteracijo. S tem omogočim izrisovanje mehkih senc.



Slika 2.1: Delovanje mapiranja senc. *Image Plane* predstavlja uporabnikov zaslon, *Shadow Map* pa sliko scene iz vidnega polja luči [13]

2.3 Druga faza: Zapis scene v medpomnilnik in injektiranje svetlobe

Takoj po izrisu senc nastopi prva faza, ki je neposredno vezana na izračun indirektne osvetlitve. V tem delu uporabljam ogljiščni in geometrijski senčilnik. Geometrijski senčilnik ima tudi vklopljeno fazo za zapis podatkov nazaj v pomnilnik (*stream output).* Ogljiščni senčilnik je standarden (izračuna 3D pozicijo v koordinatah sveta). To pozicijo prenese v geometrijski senčilnik, ki deluje na celotnih trikotnikih. GS dobi kot vhod tri ogljišča. Za vsako ogljišče izračunam osvetlitev in preberem albedo teksturo zanj (albedo tekstura predstavlja barvo materijala brez osvetlitve). Osvetlitev in albedo

združim in na koncu izračunam povprečno barvo za vsa tri ogljišča. Prav tako izračunam geometrijsko središče in normalo trikotnika.

V 32 bitno vrednost zakodiram normalo, albedo, pomnožen z osvetlitvijo, intenziteto (magnitudo albeda pomnoženega z direktno osvetlitvijo) in še zastavico za trdnost. Ta zastavica je pomembna, saj ne smemo predpostaviti, da je nek element črne barve prazen: lahko je zgolj v senci. Ta zastavica v tej fazi še ni potrebna, bistvena postane v fazi pet.

Geometrijsko središče shranim v xyz komponente 4D vektorja, zakodirano povprečno barvo, normalo, intenziteto in trdnost pa shranim v komponento w. Ta 4D vektor zapišem v pomnilnik preko faze SO.

Kodiranje opravim na sledeč način. Z enačbo 2.5 se najprej spremeni definicijsko območje vektorja normale iz [-1,1] na [0,1]. Enačba 2.6 in 2.7 zakodirata vektor normale in barve v zaporedje bitov. Operacija $a << b$ predstavlja bitni premik a za b mest v levo, $a|b$ pa bitni ali med a in b. Enačba 2.8 prikazuje kodiranje intenzitete. Na koncu pa z enačbo 2.9 združimo vse v eno 32 bitno število. Logični ali s številom 0x80000000 (zapis 0xN predstavlja številko N v šestnajstiškem sistemu) nastavi najvišji bit in s tem pove, da je ta piksel v volumetrični teksturi trden.

$$\vec{N_b} = \frac{\vec{N}}{2} + \begin{bmatrix} \frac{1}{2} \\ \frac{1}{2} \end{bmatrix} \tag{2.5}$$

$$N_p = (\vec{N_b}.x * 31)|((\vec{N_b}.y * 31) << 5)|((\vec{N_b}.z * 31) << 10) \tag{2.6}$$

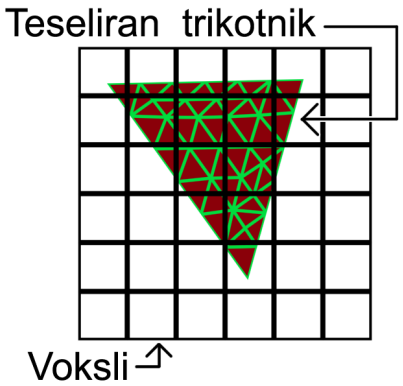
$$C_p = (\vec{C}.x * 31)|((\vec{C}.y * 31) << 4)|((\vec{C}.z * 31) << 8) \tag{2.7}$$

$$M = \|\vec{C}\| * 15 \tag{2.8}$$

$$B = (M << 27)|(N_p << 12)|C_p|0x80000000 \tag{2.9}$$

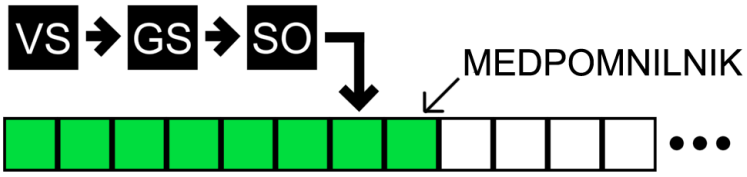
Ta način vokselizacije je zelo neučinkovit. 3D model, ki se vokselizira, mora nujno biti zelo teseliran. To pomeni, da ne smemo imeti zelo velikih

trikotnikov, temveč moramo velike trikotnike razdeliti na veliko manjših. Če poljubni trikotnik seka nek 3D piksel, bi ta piksel seveda moral dobiti barvo tega trikotnika. Problem moje metode je, da ne iščem preseka s 3D piksli, temveč gledam, ali v določeni 3D piksel pade center nekega trikotnika. Če nek trikotnik samo seka 3D piksel, ampak nima središča v njem, potem algoritem sploh ne bo obarval 3D piksla. Zaradi tega imajo vsi 3D modeli veliko več ogljišč, kot pa je potrebno.



Slika 2.2: Prikaz originalnega trikotnika (rdeče), teseliranega trikotnika (zeleno) in 2D mreže vokslov.

Za sceno uporabljam Dabrovicov 3D model atrija palače Sponze, ki se nahaja v Dubrovniku. 3D model Sponze ima 354628 ogljišč, 3D model dodatkov za Sponzo (stebri, zavese ...) 59228, 3D model povezanih krogel pa jih ima 2394. Veliko število trikotnikov povzroči dve težavi: prva je zelo očitna, in sicer gre za veliko količino ogljišč, ki jih mora GPE obdelati v cevovodni fazi VS in GS. Prav tako to povzroči veliko večjo število podatkov zapisanih v medpomnilnik preko faze SO. Drug problem pa se pojavi pri trikotnikih, ki so veliki manj kot kvadrat 2x2 piksla na ekranu, potem ko so že izrisani. Zaradi tega se PS faza cevovoda izvede po nepotrebnem za veliko število pikslov. Ta problem pride povsem iz strojne arhitekture GPE.



Slika 2.3: Potek podatkov skozi cevovod v drugi fazi. Na koncu se iz GS preko SO piše v medpomnilnik.

2.4 Tretja faza: izris scene

V tej fazi izrišemo dejansko vidno sceno. V tej fazi se uporabita senčilnika VS in PS. VS je standarden (transformacija za svet, kamero in projekcijo). V PS se izračuna osvetlitev za reflektorsko luč, ugotovi se, ali je objekt v senci. Na koncu celotno sceno zapišem v tri teksture: v prvo zapišem končno barvo (albedo pomnožen z direktno osvetlitvijo), v drugo normale in v tretjo samo albedo.

2.5 Četrta faza: vokselizacija

Sedaj izvedemo prepis generiranega seznama iz faze 2 v volumetrično teksturo, ki je velika 128 kubičnih pikslov v vsako dimenzijo. Za ta postopek uporabim splošno namenski senčilnik (*compute shader).* Ta senčilnik, v gručah po 64 niti, izvede prepis v volumetrično teksturo (razlog za 64 je v tem, da grafična strojna oprema ponavadi izvaja senčilnike v gručah po 64 ali vsaj večkratniku 64).

Trenutno 3D tekstura pokriva področje (kocko) veliko 40x40x40 enot (torej 20 enot na vsako stran iz koordinatnega izhodišča). Vse, kar je zunaj te kocke, se ne bo štelo kot vir indirektne osvetlitve.

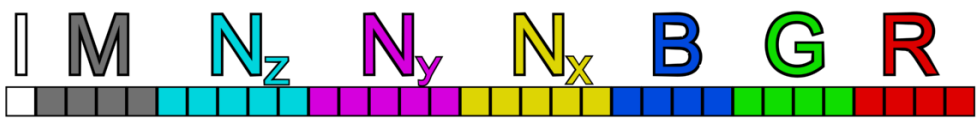
Spodnja enačba prikazuje pretvorbo koordinat sveta (ki so v razponu

[-20,20]) v koordinate volumetrične teksture (razpon [0,128]).

$$\vec{T} = (\frac{\frac{\vec{W}}{20}}{2} + \begin{bmatrix} 0.5 \\ 0.5 \\ 0.5 \end{bmatrix}) * 128 \tag{2.10}$$

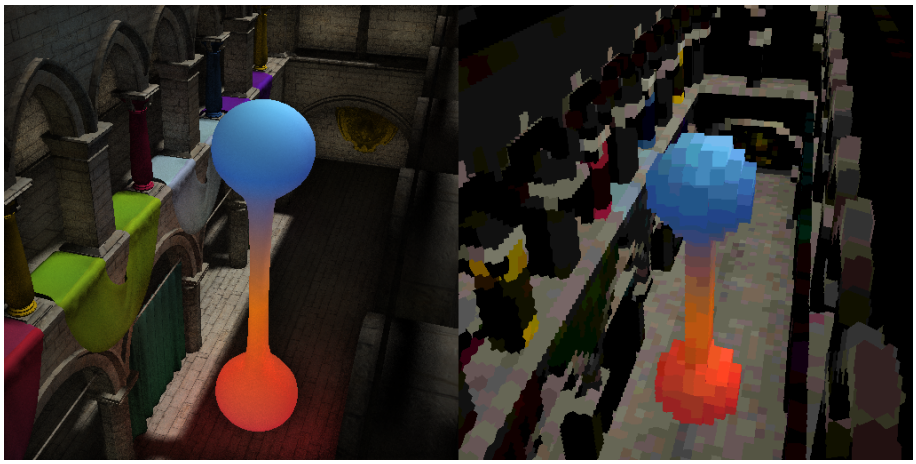
Za prepis enega elementa se najprej dekompresira komponenta w 4D vektorja, tako da dobimo barvo in normalo. Iz komponent xyz preberemo pozicijo in jo mapiramo iz koordinat sveta v koordinate teksture (razpon 0 do 128). Tukaj potem pridemo do hude težave glede učinkovitosti: v določen volumetrični piksel lahko pade več kot en trikotnik. To pomeni, da GPE piše v isti piksel (iste pomnilniške lokacije) iz več niti. To povzroči nedeterministične rezultate, saj vrstni red zapisa ni definiran. Ta se vizualno kaže kot utripanje 3D pisklov. Ta problem lahko v Direct3D 11 rešimo z uporabo atomičnih funkcij (te funkcije zagotavljajo, da bo v neko pomnilniško lokacijo pisala samo ena nit hkrati). Seveda ravno to uniči celotno učinkovitost GPE, saj se morajo niti med seboj čakati. A tudi če uporabimo sinhronizirano pisanje, nam brez dodatnih sprememb, to ne bo dosti pomagalo: moramo nekako določiti pogoj, kakšno vrednost bomo zapisali v volumetrično teksturo. V primeru, da se v en 3D piksel pišeta zelen in en rdeč trikotnik, je vprašanje, kakšne barve naj bo ta 3D piksel. Preprost odgovor je, da rezultat povprečimo. Tega na žalost ne moremo narediti na enostaven način.

Problem je, da ne moremo vedeti, kakšne barve so ostali trikotniki, ki so bodo zapisali v to teksturo. Ne moremo namreč kar prebrati že zapisano vrednost v 3D pikslu, saj nimamo zagotovila, da so ostale niti že zapisale svojo vrednosti. Problem sem rešil tako, da sem uporabil HLSL funkcijo **InterlockedMax**. Ta funkcija primerja 32-bitne celoštevilske tipe in sinhronizirano shrani največjega (torej uporabi obstoječega ali pa zamenja obstoječo vrednost z novo, če je večja). V fazi GS sem pri kodiranju normale in barve v bitih 28 do 31 shranil še intenziteto barve (torej magnitudo vektorja osvetlitve, pomnoženega z albedom). S pomočjo te zakodirane intenzitete vedno zapišem najsvetlejši trikotnik v 3D piksel in se tako izognem utripanju v skoraj vseh primerih. Implementiral sem tudi enostaven izrisovalec



Slika 2.4: Kodiranje barve (R, G, B), normale (N_x, N_y, N_z), intenzitete (M) in trdnosti (I). Vsak kvadratek predstavlja en bit, vseh skupaj je 32.

volumetričnih tekstur. Ta je bil v ogromno pomoč na začetku, saj je na grafični kartici v veliko primerih težko (oz. celo nemogoče) razhroščevati. Te faze sploh nisem imel možnosti razhroščevati, saj orodje GPUPerfStudio (razhroščevalnik za GPE podjetja AMD) v mojem primeru ni podpiralo razhroščevanje CS.



Slika 2.5: Na levi vidimo dejanski rezutat algoritma, na desni pa vokselizirano sceno v 3D teksturi (kameri nista na povsem enaki poziciji).

2.6 Peta faza: simulacija odboja svetlobe

2.6.1 Priprava za algoritem

Sedaj imamo vse pripravljeno za dejanski izračun indirektne osvetlitve. Najprej izrišem kvadrat (dva trikotnika), ki pokriva celotno vidno površino. Sedaj za vsak piksel v tem kvadratu poženemo PS, ki opravi simulacijo. Najprej preberemo normalo in globino za trenutni piksel. Nato s pomočjo globine in inverzne matrike kamera-projekcija izračunamo 3D pozicijo v koordinatah sveta. Temu sledi izračun velikosti enega 3D piksla v koordinatah sveta.

2.6.2 Streljanje žarkov

Srce algoritma je streljanje žarkov in ugotavljanje presečišča žarka z volumetričnim pikslom v 3D teksturi. Postopek poteka sledeče: v stožcu izstrelim 30 žarkov. Konica stožca je na 3D koordinatah sveta za trenutni piksel. Stožec je usmerjen v smeri normale na površini, ki je na trenutnem pikslu. Za vsako iteracijo izračunam naključni 3D vektor, ki ga uteženega prištejem k vektorju normale. Ta utež določa radij stožca, če je dovolj velika, pa se stožec spremeni v polkroglo. Zaradi uteži se lahko zgodi, da ta vektor kaže v povsem nasprotno smer kot vektor normale. To enostavno preverim s skalarnim produktom in če je ta manjši od 0, pomnožim vektor z -1.

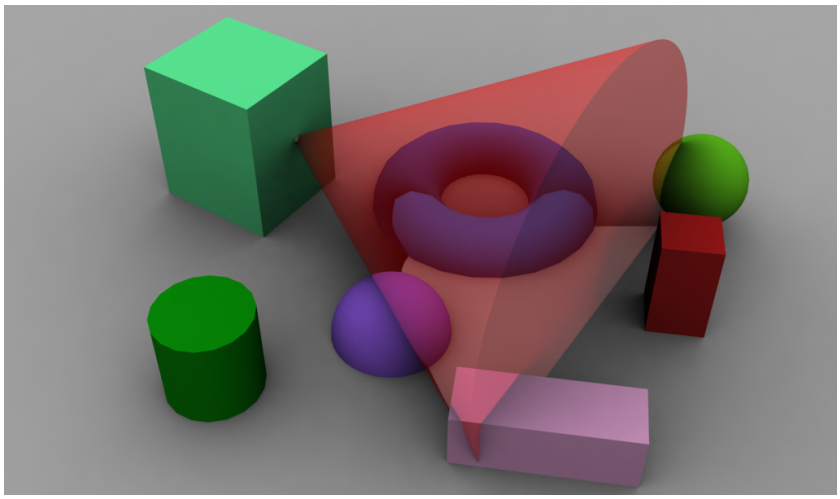
Dodaten problem predstavlja izvor žarka (torej konica stožca). Če namreč žarek izstrelimo iz enakih koordinat sveta, kot jih ima trenutni piksel, se lahko zgodi, da bo zadel volumetrični piksel, ki predstavlja to površino, iz katere se je žarek ustrelil. Ta problem lahko odpravimo na dva načina. Pri prvem izvor stožca premaknemo v smeri normale vsaj za velikost enega 3D piksla, drugi način pa je, da žarku določimo minimalno prepotovano pot. Torej če žarek zadane 3D piksel še preden prepotuje dolžino enega 3D piksla, ta zadetek ignoriramo. V delu uporabljam drugo metodo, saj v primeru bližnjih 3D pikslov daje lepše rezultate.

$$\vec{D} = \frac{\vec{N} + \vec{R}}{\|\vec{N} + \vec{R}\|} \tag{2.11}$$

$$\vec{A_0} = \vec{W} \tag{2.12}$$

$$\vec{A_n} = \vec{A_{n-1}} + \vec{D}t \tag{2.13}$$

Z enačbo 2.11 normali površine dodamo naključen vektor $\vec{R}$ in rezultat normaliziramo. $\vec{D}$ sedaj predstavlja smer žarka. Nato z 2.12 določimo začetni položaj žarka tako, da uporabimo 3D pozicijo površine $\vec{W}$ na trenutnem pikslu v koordinatah sveta. Nato iteriramo z uporabo 2.13. Tu je t velikost koraka. Iteriramo, dokler ne pridemo do maksimalnega števila iteracij ali dokler ne zadanemo trden 3D piksel.

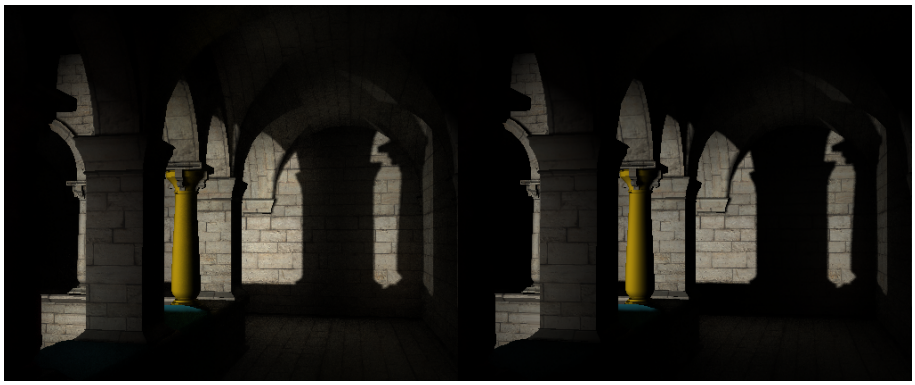


Slika 2.6: Žarki se streljajo iz vrha stožca v naključni smeri, a so omejeni z dnom stožca (dno ima dejansko obliko dela krogle, ki ima središče v vrhu stožca).

Uporabljam zanko z maksimalno 60 iteracijami. Vsako iteracijo žarek prepotuje določen korak po poti. Če je ta korak velik, bi sicer lahko uporabil veliko manj iteracij, a bi potem lahko zgrešil določene 3D piksle. Potrebno

je torej ujeti dovolj velik korak, da žarek prepotuje dovolj dolgo pot in da ne zgreši kakšnega 3D piksla. Če žarek namreč ne prepotuje dovolj dolge poti, se lahko zgodi, da ne vidimo odboja svetlobe iz bolj oddaljenih površin. Vsako iteracijo korakanja po poti pretvorimo 3D koordinate sveta žarka v koordinate volumetrične teksture in preberemo 3D piksel iz tiste lokacije. Če ugotovimo, da je trenutni 3D piksel poln, vrnemo njegovo barvo, ki jo pomnožimo z obratno vrednostjo kvadrate oddaljenosti od površine, ki je izsevala to svetlobo. Razlog za to je v tem, da intenziteta svetlobe pada s kvadratom oddaljanosti od izvora svetlobe. Če je zadeti 3D piksel prazen nadaljujemo s korakanjem po poti. Če pridemo do konca zanke in ne zadanemo ničesar, vrnemo črno barvo.

Sedaj lahko ta končni rezultat, preden ga vrnemo, pomnožimo še z geometrijskim faktorjem. To je skalarni produkt med normalo površine, ki odbija svetlobo in normalo površine, ki prejema odbito svetlobo. Seveda to ni popolnoma fizikalno pravilno, saj bi moral upoštevati vpadni vektor svetlobe na površino in potem odbiti to svetlobo glede nanj in na normalo površine. Ampak s to metodo to ni mogoče učinkovito narediti za več kot eno luč. Ali se bo geometrijski faktor uporabljal ali ne je odvisno od potreb uporabnika. Menim, da ga je najbolje ignorirati, saj preveč potemni sliko, ker nimamo večkratnih odbojev svetlobe.



Slika 2.7: Na levi je scena brez geometrijskega faktorja, na desni pa z njim. Najbolj očitna razlika je svetlosti sence stebra.

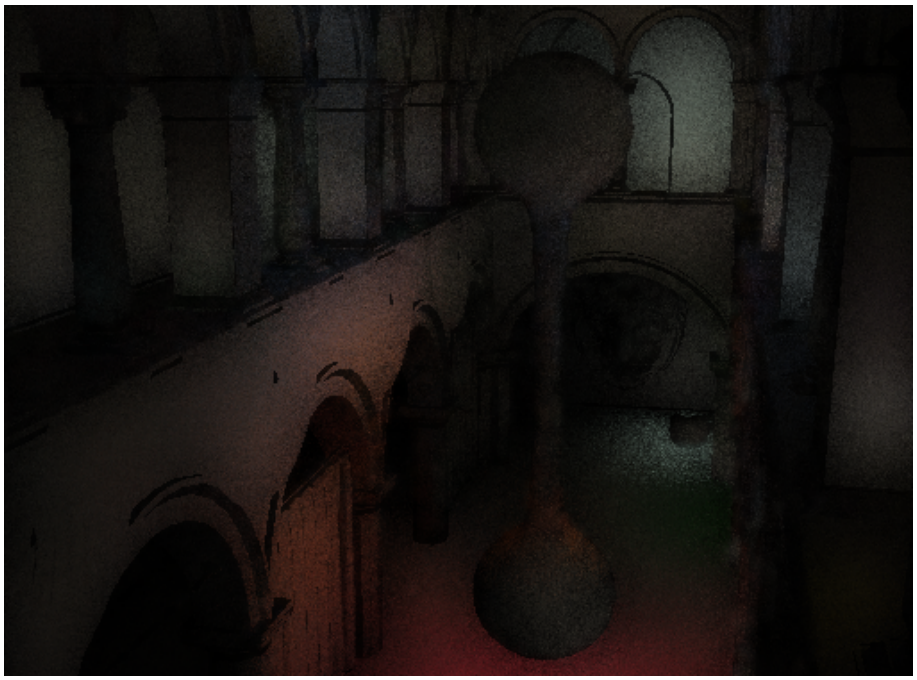
Ko imamo rezultate za vsak izstreljeni žarek, jih seštejmo in utežimo s številom iteracij. To seveda ni fizikalno pravilno, saj bi moral upoštevati površino dna stožca, a menim, da v tem delu to ni bistvenega pomena. Na koncu rezultat shranimo v teksturo. Ta korak se opravi pri eni četrtini dejanske velikosti površine, na katero se izriše scena. To ima ogromen pozitiven vpliv na hitrost delovanja in zanemarljiv vpliv na izgled, saj je indirektna osvetlitev zelo nizko frekvenčen pojav, torej se ne razlikuje zelo na sosednjih pikslih (če seveda tam ni robu).

Sledi del kode, ki sledi žarku v določeni smeri skozi volumetrično teksturo in ugotavlja zadetek:

```
float3 shootRay(float3 normalizedViewDirection,float3
   rayOriginPosition,float stepSize,float3
   rayOriginSurfaceNormal,float textureSize)
{
  int numIter=60;
  float3 viewDir = normalizedViewDirection;
  float3 currentPos = rayOriginPosition;
  float3 voxelNormal = float3(0,0,0);
  float3 texSample = float3(0,0,0);
  float distanceTraveled = 0.0f;
  float maxRange = stepSize * (float)numIter;
  float invMaxRange = 1.0f/maxRange;

  [loop]for(int i=0;i<numIter;++i)
  {
    // pretvorimo 3D koordinate sveta v razpon [0,1],
    float3 normalizedVolumeCoords =
       posToNormalizedVolumeTexcoords(currentPos);

    // preberemo 3D piksel iz pomnilnika
    // glede na trenutne koordinate
    uint tex3DSample = texGIVolume.Load(int4(
       normalizedVolumeCoords * textureSize,0));
```

```
    // dekompresiramo prebrani 3D piksel
    uint isSolid =
        unpackWithNormalsAndMagnitudeAndHighestBitForSolid(
        tex3DSample,voxelNormal,texSample);

    // geometrijski faktor
    float dp = saturate(dot(-voxelNormal,
        normalizedViewDirection));

    distanceTraveled += stepSize;

    // premaknemo zarek za en korak v njegovo smer
    currentPos += viewDir * stepSize;

    if( isSolid > 0 && distanceTraveled >= 1.1f )
    {
      //padec intenzite z oddaljenostjo
      float distanceAttenuation = 1.0f - distanceTraveled *
          invMaxRange;

      // tu lahko odstranim ' * dp'
      return texSample.xyz * distanceAttenuation *
          distanceAttenuation * dp;
    }
  }

  // ce nic ne zadanemo vrnemo crno barvo
  return float3(0,0,0);
}
```

Programska koda 2.1: Premik žarka po 3D teksturi in ugotavljanje zadeteka 3D piksla.



Slika 2.8: Rezultat 5. faze (že povečan in zamegljen).

2.7 Šesta faza: zlitje direktne in indirektne osvetlitve

Sledi še zadnja faza v celotnem postopku. V tretji fazi smo izrisali sceno z direktno osvetlitvijo v teksturo, ki ji sedaj uteženo prištejemo teksturi iz četrtega koraka. Ta je seveda štirikrat manjša, zato jo moramo povečati. Pri povečanju jo tudi zameglim z upoštevanjem robov, saj bi se drugače lahko pojavili žareči predeli ob robovih. Utež za seštevanje teksture bi bila 1, če bi hoteli upoštevati zakone fizike. A v igrah se mnogokrat pojavi potreba po posebnem vizualnem slogu, zato je dobro, da tu uporabimo utež, ki jo lahko uporabnik nadzoruje. Spodnja enačba prikazuje način zlivanja. $\vec{K}$ je končni rezultat, $\vec{D}$ je scena samo z direktno osvetlitvijo, $\vec{A}$ je albedo, $\vec{I}$ je (povečan in zamegljen) rezultat faze 5 in a je utež.

$$\vec{K} = \vec{D} + \vec{A}\vec{I}a \tag{2.14}$$

2.8 Meritve porabe časa

Sledeče meritve so bile opravljene na računalniku s CPE Core i5, 4GB sistemskega pomnilnika in GPE AMD Radeon 6950 (2GB grafičnega pomnilnika). V tabeli 5.1 vidimo rezultate.

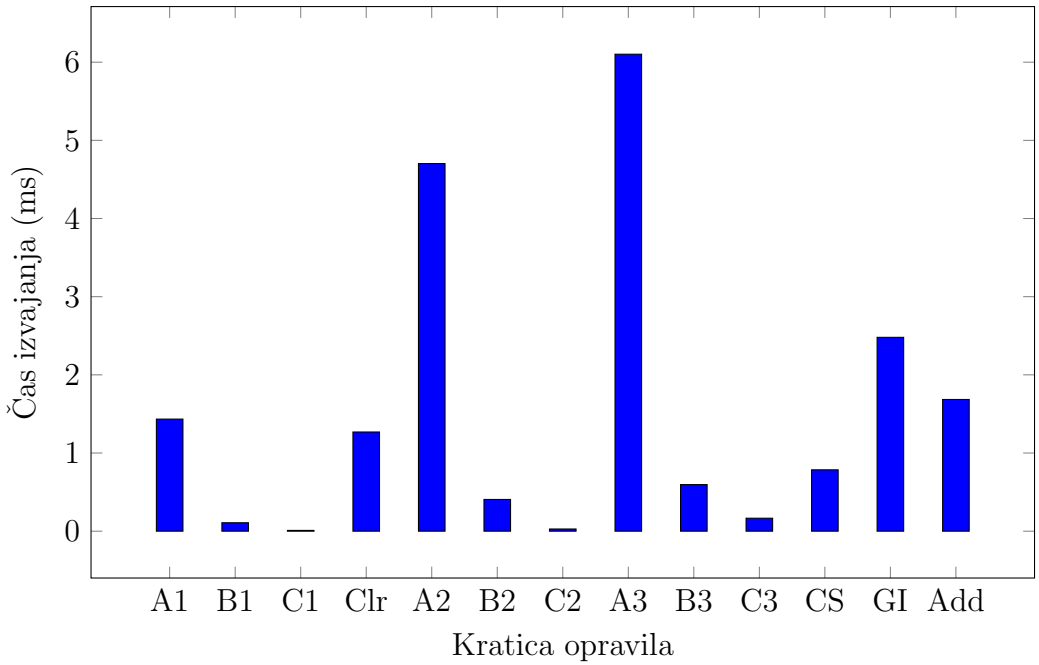


Tabela 2.1: Graf porabe časa

Oznaka	Pomen
A_n	3D model Sponze v fazi n
B_n	3D model dodatkov za Sponzo v fazi n
C_n	3D model vrtečih povezanih krogel v fazi n
Clr	Čiščenje medpomnilnika za GS rezultat
CS	Prepis GS rezultata iz medpomnilnika v volumetrično teksturo
GI	Sledenje žarkom po volumetrični teksturi
Add	Zlitje direktne in indirektne osvetlitve

Tabela 2.2: Legenda za graf porabe časa

Najbolj opazen je 3D model Sponze z oznakami A_n. Razloga za to sta dva: prvi je veliko število ogljišč in primitiven algoritem za vokselizacijo, kar zelo obremeni fazo VS in povzroči v fazi GS za vsak trikotnik eno sinhronizacijo. Razlog za ogromno porabljenega časa v A_3 je zaradi izrisa senc, saj uporabljajo zelo primitiven algoritem za mehčanje senc. V splošnem lahko rezultate faze 3 (torej A_3, B_3, C_3) ignoriramo. Ta faza namreč ni bistvo tega dela in se jo da zelo optimizirati. Enako velja za fazo 1. Za končno porabo časa se morajo upoštevati faza 2 (A_2, B_2, C_2), *Clr*, *GS*, *GI* in *Add*. To znaša 11,356ms. Spomnimo se, da je meja za 60 slik na sekundo 16.6 ms, za 30 slik na sekundo pa 33.3 ms.

2.9 Podporna tehnologija za realizacijo

2.9.1 Visokonivojski pregled podporne tehnologije

Za vizualizacijo scene najprej potrebujemo program, ki teče na CPE. Ko imamo to postavljeno, lahko napišemo še programe, ki se izvajajo na GPE, to so tako imenovani *senčilniki* (to ni vezano na sence). Naslednji nujen del so seveda 3D modeli objektov, ki sestavljajo sceno. 3D model vsebuje vse geometrijske podatke, ki so potrebni za izris nekega objekta v virtualnem svetu (najbolj pogosto so to 3D koordinate pozicije ogljišč, normale površin in koordinate za teksturo).Na 3D modele nalepimo sliko (teksturo), ki jim da bolj realen izgled. Na koncu vse skupaj izrišemo v teksturo, ki jo potem prikažemo na zaslon.

Od mnogih prejšnjih projektov se mi je nabralo že kar precej podporne tehnologije, ki jo uproabljam v diplomskem delu. Lahko bi rekli, da se mi je oblikoval že zelo preprost pogon za igre. Program je ustvarjen za operacijski sistem Windows. Za dostop do grafične kartice uporabljam standard Direct3D 11. Najprej ustvarim okno, v katerega se scena izrisuje, nato inicializiram Direct3D. Temu sledi nalaganje 3D modelov in tekstur. Ko je vsa inicializacija končana, začnem izrisovati sceno. Sproti tudi lovim uporabnikove pritiske na tipke in miško.

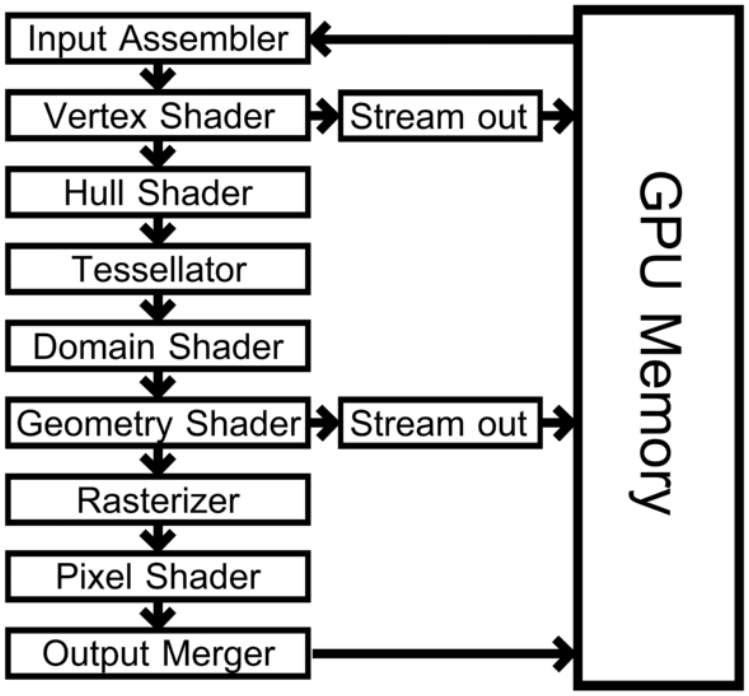
2.9.2 Skupni modul

Srce pogona je skupni modul (projekt *N*XCore v Visual Studiu). V njem so zajete naslednje pomembnejše stvari: definicije podatkovnih tipov, C++ definicije (*defines*), pogoste podatkovne strukture (seznam, vrsta ...), pomočniki za delo z datotekami in grafi, matematični pripomočki (vektorji, matrike ...), časovnik in še razne ostale pogosto uporabljene funkcije. Vsi ostali moduli referencirajo ta modul.

2.9.3 Grafični moduli

V pogonu imam podporo za Direct3D in OpenGL, a imam osebno preferenco do Direct3D, zato je OpenGL podprt samo za najbolj enostavne stvari. Modul za Direct3D, *NXD3D* vsebuje pomočnike za inicializacijo Direct3D, prevajanje, shranjevanje in nalaganje že prevedenih senčilnikov, za nalaganje in upravljanje s teksturami in ostalimi Direct3D objekti. Podpira tudi avtomatsko generiranje objektov *input layout*. Modul *NXD3D* se uporablja v modulu *NXRenderer*, ki je višje nivojska abstrakcija Direct3D. Ima menedžerje za upravljanje z Direct3D objekti in je sposobnen naložiti 3D model s teksturami iz datoteke. Ta modul upravlja z izrisovalnimi objekti (en izrisovalni objekt predstavlja C++ razred *Renderable*).

Direct3D deluje na principu stanj: ko se eno stanje nastavi, potem to stanje ostane, dokler ne nastavimo drugega. Nastavljanje stanj zahteva klic funkcije operacijskega sistema, potem mora iti skozi gonilnik (ki je razdeljen na uporabniški in sistemski del) in na koncu se morata v nekaterih primerih CPE in GPE sinhronizirati. Ves ta postopek traja kar dolgo, tako da je dobro, da ne nastavljamo stanj po nepotrebnem. Ta modul si lahko shrani stanja, ki so bila nastavljena, in jih sortira na tak način, da je potrebnih čim manj klicev do knjižnice Direct3D. S tem se prihrani CPE čas.



Slika 2.9: Direct3D 11 cevovod.

2.9.4 Modul za nalaganje 3D modelov

Objekte, prikazane v sceni, moramo seveda naložiti iz zunanjih datotek. Za osnoven 3D model morajo v datoteki obstajati podatki o poziciji oglišč, normale ogljišč (vektor, ki se uporabi pri računanju osvetlitve) in koordinate UV (te povejo, kako se slika nalepi na trikotnik). Ponavadi shranimo tudi seznam, ki pove, katera tri ogljišča se morajo povezati med seboj, da dobimo trikotnik. Slednjemu seznamu se reče indeksni seznam in se dandanes uporablja vedno, ko imamo kompleksen 3D model z večjim številom ogljišč. Če ne bi imeli indeksnega seznama, bi bila mnoga ogljišča podvojena, saj se ogljišča ne bi morala deliti s sosednjimi trikotniki.

Ali potem vse zgoraj naštete podatke zapišemo kot tekstovno ali binarno datoteko, je odvisno od uporabnikovih potreb. Tekstovne datoteke so bolj fleksibilne, ampak so zelo počasne za branje v primerjavi z binarnimi. Za moj pogon 3D modele iz zunanjih 3D urejevalnikov (3D Studio Max, Blen-

der ...) shranim v format *Collada* (končnica formata je *DAE*). Ta format je tekstovni v zapisu XML. Za branje in izluščenje podatkov o 3D modelu in teksturah uporabljam knjižnico *ASSIMP* (*Open Asset Import Library*) [12].

Ker je branje velikih *DAE* datotek zelo počasno sem ustvaril dva preprosta formata za shranjevanje informacij o 3D modelu: to sta formata *NXMESH* in *NXMAT*. *NXMESH* shranjuje v binarnem formatu vse podatke o geometriji, *NXMAT* pa podatke o teksturah. NXMESH format shranjuje podatke ločeno, torej podatki o poziciji so posebej, podatki o normalah so pravtako posebej itd. Ker je to binarni format, se ga lahko prebere veliko hitreje kot pa DAE. Ravno zaradi tega se mi avtomatsko vsaka prebrana *DAE* datoteko pretvori v par *NXMESH* in *NXMAT* datotek. Naslednjič, ko želimo prebrati DAE datoteko, preverim ali *NXMESH* in *NXMAT* datoteki obstajata. Če obstaja, uporabim te, saj je časovni prihranek ogromen.

2.9.5 Modul za pomoč pri uporabi operacijskega sistema

Modul, ki ga predstavlja projekt *NXWindowsCore*, ima implementirane razrede za enostavno ustvarjanje oken za operacijski sistem *Windows* in omogoča uporabniku pridobivanje podatkov o uporabnikovem vnosu preko tipkovnice in miške.

2.9.6 Izvajalni modul

Vsi potrebni moduli so skupaj povezani v izvajalnem modulu. Ta Visual Studio projekt se imenuje *FrameworkV1* in ta se na koncu prevede in poveže v izvršilno datoteko (končnica *exe*), ki jo uporabnik potem zažene. Metoda, ki jo predstavljam v tem delu, je implementirana v tem modulu.

2.9.7 Koordinatni sistemi

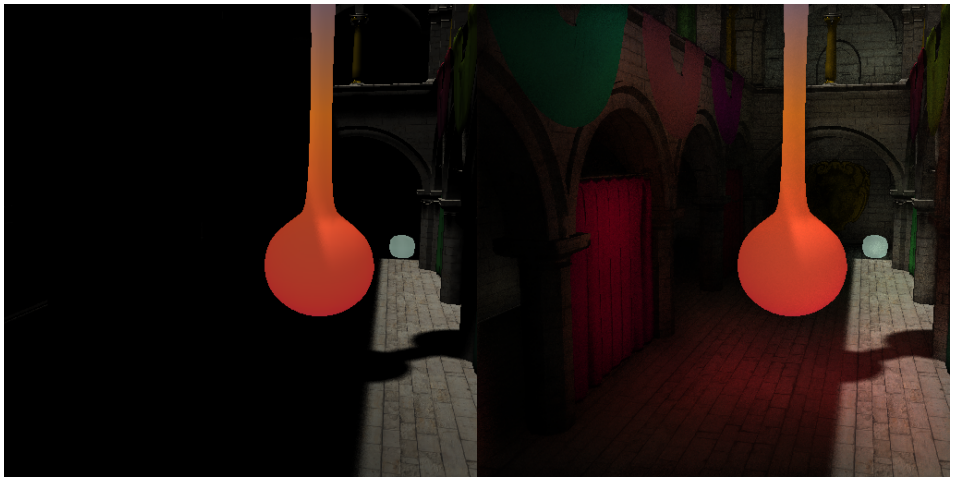
Kadarkoli se ukvarjamo z vektorji, jih moramo postaviti v neki kontekst. To storimo tako, da nekje določimo koordinatno izhodišče in osi (skoraj vedno

je to ortonormirana baza).

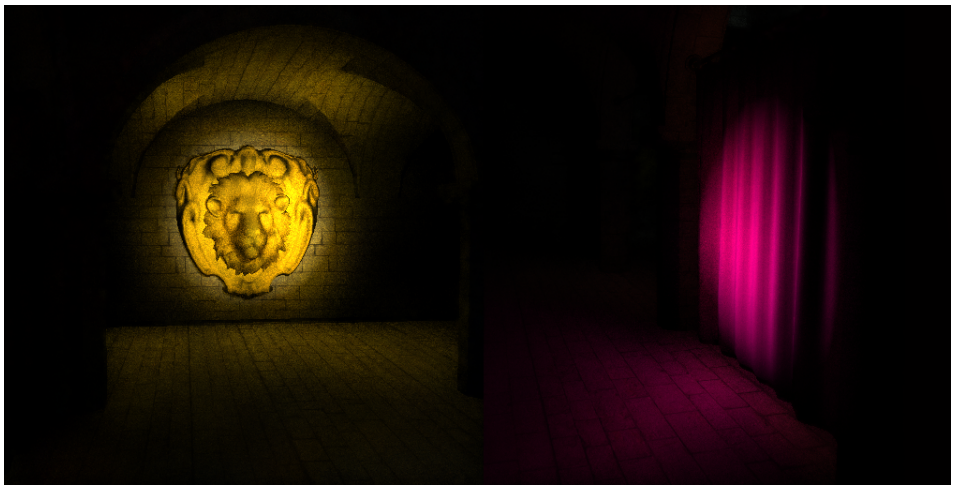
V računalniški grafiki je zelo priročno uporabljati več različnih koordinatnih sistemov. Najprej objekt postavimo v svet. Temu pravimo koordinatni sistem sveta. Potem ustvarimo koordinatni sistem iz pogleda kamere. To naredimo tako, da sestavimo 4x3 ali 4x4 matriko. Najbolj leve tri komponente vrstic 1, 2 in 3 predstavljajo bazne vektorje kamere (vektor, ki gleda desno od kamere, vektor, ki je usmerjen navzgor relativno na kamero, in vektor v smeri pogleda). V četrto vrstico postavimo x,y in z komponente translacije. Sedaj izračunamo inverz te matrike in jo uporabimo kot matriko, ki pretvori točke in vektorje iz koordinatnega sistema sveta v koordinatni sistem pogleda. Ustvarimo še projekcijsko matriko, ki pretvori točke in vektorje iz koordinatnega sistema pogleda v projekcijski koordinatni sistem. Ta se na koncu avtomatsko pretvori v koordinatni sistem slike.

2.10 Primeri

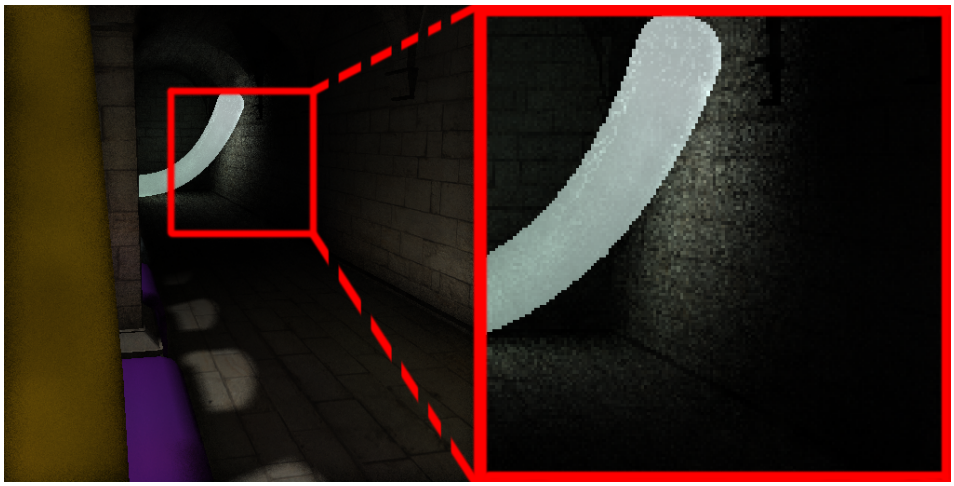
Slika 2.10: Primer 1: Vidimo odboj svetlobe od tal na bližnji zid in stebre. V daljavi celotna modra cev seva svetlobo. Levo brez, desno pa z indirektno osvetlitvijo.



Slika 2.11: Primer 2: Tukaj vidimo 3D model vrtečih krogel, ki seva svetlobo. Brez indirektne osvetlitve je praktično nemogoče uporabiti kar nespremenjen 3D model kot izvor svetlobe. Levo brez, desno pa z indirektno osvetlitvijo.



Slika 2.12: Primer 3: Na teh dveh slikah se dobro vidi obarvana indirektna osvetlitev, na levi rumena od leva, na desni pa roza od zavese.



Slika 2.13: Primer 4: V daljavi vidimo šum. Da bi se ga znebil, bi moral še izboljšati algoritem za povečanje in megljenje teksture iz faze 4.

Poglavje 3

Zaključek

Zagotovo bodo na trg kmalu prišle igre, ki bodo uporabljale podoben algoritem za indirektno osvetlitev. Podpora za *Voxel Cone Tracing* bo kmalu v pogonu Unity. Iz pogona Unreal Engine 4 so to metodo sicer odstranili, saj menijo, da je prepočasna.

Videli smo, kako lahko s pomočjo povsem drugačno predstavitvijo scene, v mojem primeru z vokselizirano, omogočimo učinkovito računanje indirektne osvetlitve. Ob računanju direktne osvetlitve lahko sledimo žarkom skozi vokselizirano sceno in dobimo zelo dober približek indirektne osvetlitve.

Moja metoda deluje veliko bolje, kot sem pričakoval. Rezultati seveda niso popolni v nekaterih primerih, a na splošno gledano je skok v realnosti virtualne scene ogromen. Največje presenečenje zame je bilo v hitrosti izvajanja četrte faze, to je sledenje žarkom po volumetrične teksturi. Bil sem trdno prepričan, da bo prav ta faza najbolj počasna. Na koncu se je izkazalo, da je največji problem v učinkoviti vokselizaciji. V prihodnosti bi svoji implementaciji najprej izboljšal fazo vokselizacije. Ostale faze bi za večji skok v učinkovitosti morale delovati na popolno drugačen način v primeru, da bi hotel dobiti tako dramatičen skok v hitrosti, kot pa če bi izboljšal zgolj fazo vokselizacije.

Literatura

[1] Eric Lengyel
"Mathematics for 3D Game Programming and Computer Graphics, Third Edition".

[2] Mickael Gilabert, Nikolay Stefanov,
"Deferred Radiance Transfer Volumes".
http://www.gdcvault.com/play/1015326/Deferred-Radiance-Transfer-Volumes-Globals

[3] Cyril Crassin, Fabrice Neyret, Miguel Sainz, Simon Green, Elmar Eisemann,
"Interactive Indirect Illumination Using Voxel Cone Tracing".
https://research.nvidia.com/sites/default/files/publications/GIVoxels-pg2011-authors.pdf

[4] Anton Kaplanyan, Carsten Dachsbachery,
"Cascaded Light Propagation Volumes for Real-Time Indirect Illumination".
http://crytek.com/sites/default/files/20100301_lpv.pdf

[5] "Transformers: Dark of the Moon".
http://en.wikipedia.org/wiki/Transformers:_Dark_of_the_Moon

[6] "They Want To Make Space Combat Games Fun Again".
http://www.kotaku.com.au/2011/07/they-want-to-make-space-combat-games-fun-again/

[7] "The Review Quake 3 Arena".
http://legacy.macnn.com/thereview/reviews/quake3/quake3.shtml

[8] Crysis slika.
http://www.neogaf.com/forum/showthread.php?p=48122687

[9] OpenGL.
http://en.wikipedia.org/wiki/OpenGL#OpenGL_1.3

[10] Direct3D.
http://en.wikipedia.org/wiki/Microsoft_Direct3D

[11] Slika mapiranja luči.
http://3dgep.com/?tag=lightmapping

[12] ASSIMP C++ knjižnica.
http://assimp.sourceforge.net/main_downloads.html

[13] Shema delovanja mapiranja senc.
http://http.developer.nvidia.com/CgTutorial/cg_tutorial_chapter09.html

[14] Spletni forum na katerem sem skozi leta pridobil ogromno informacij.
http://www.gamedev.net/index

[15] Spletni forum na katerem sem skozi leta pridobil ogromno informacij.
http://www.gamedev.net/index

[16] "Graphics processing unit"
http://en.wikipedia.org/wiki/Graphics_processing_unit