

UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Marko Tatić

Odloženo upodabljanje

DIPLOMSKO DELO NA UNIVERZITETNEM ŠTUDIJU

MENTOR: doc. dr. Matija Marolt

Ljubljana, 2013

Rezultati diplomskega dela so intelektualna lastnina avtorja in Fakultete za računalništvo in informatiko Univerze v Ljubljani. Za objavljane ali izkoriščanje rezultatov diplomskega dela je potrebno pisno soglasje avtorja, Fakultete za računalništvo in informatiko ter mentorja.

Besedilo je oblikovano z urejevalnikom besedil $\text{\LaTeX}$.



Št. naloge: 01948 / 2013
Datum: 4.9.2013

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

Kandidat: **MARKO TATIĆ**

Naslov: **ODLOŽENO UPODABLJANJE
DEFERRED RENDERING**

Vrsta naloge: DIPLOMSKO DELO UNIVERZITETNEGA ŠTUDIJA

Tematika naloge:

V diplomskem delu raziščite področje realno-časovnih rešitev upodabljanja. Osredotočite se na postopke odloženega upodabljanja, predvsem na odloženo senčenje in odloženo osvetljevanje. Razvijte svojo rešitev za upodabljanje in opišite izdelavo potrebnih medpomnilnikov, postopek izrisa geometrije, postopek osvetljevanja ter izdelavo geometrije za volumne luči.

Mentor:

doc. dr. Matija Marolt



Dekan:

prof. dr. Nikolaj Zimic

IZJAVA O AVTORSTVU DIPLOMSKEGA DELA

Spodaj podpisani Marko Tatić, z vpisno številko **63060064**, sem avtor diplomskega dela z naslovom:

Odloženo upodabljanje

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 9. decembra 2013

Podpis avtorja:

Rad bi se zahvalil mentorju za nasvete pri izdelavi diplomskega dela. Zahvaljujem se tudi družini za podporo v času študija.

Kazalo

1	Uvod	1
1.1	Takojšnje upodabljanje	3
2	Odloženo upodabljanje	5
2.1	Prednosti odloženega upodabljanja	8
2.2	Slabosti odloženega upodabljanja	8
2.3	Pristopi k odloženemu upodabljanju	9
3	Razvojno okolje	13
4	Razvoj rešitve	15
4.1	Izdelava geometričnega medpomnilnika	15
4.2	Izris geometrije in materialnih lastnosti	17
4.3	Izdelava osvetlitvenega medpomnilnika	19
4.4	Izdelava geometrije volumnov luči	20
4.5	Izris osvetlitve	23
4.6	Končna osvetlitev	35
5	Sklepne ugotovitve	37
A	Slike nastale med razvojem	39

Povzetek

Upodabljanje je postopek pretvorbe 3D objektov v 2D sliko primerno za prikaz na zaslonu. V diplomskem delu obravnavamo področje realno-časovnih rešitev upodabljanja, s poudarkom na postopku odloženega upodabljanja. Predstavimo klasičen pristop takojšnjega upodabljanja ter dva najbolj tipična pristopa odloženega upodabljanja: odloženo senčenje in odloženo osvetljevanje. V nadaljevanju razvijemo svojo rešitev odloženega upodabljanja s pomočjo programskih jezikov *C#* in *HLSL* ter ogrodja *XNA*. Opišemo izdelavo potrebnih medpomnilnikov, postopek izrisa geometrije, postopek osvetljevanja ter izdelavo geometrije za volumne luči. Poleg klasičnih tipov luči pa razvijemo tudi svojo rešitev za neonsko luč, ki ponuja zanimive možnosti osvetljevanja. Za konec predstavimo nekaj možnih izboljšav.

Ključne besede: upodabljanje, odloženo upodabljanje, odloženo senčenje, odloženo osvetljevanje, neonska luč, *C#*, *HLSL*, *XNA*.

Abstract

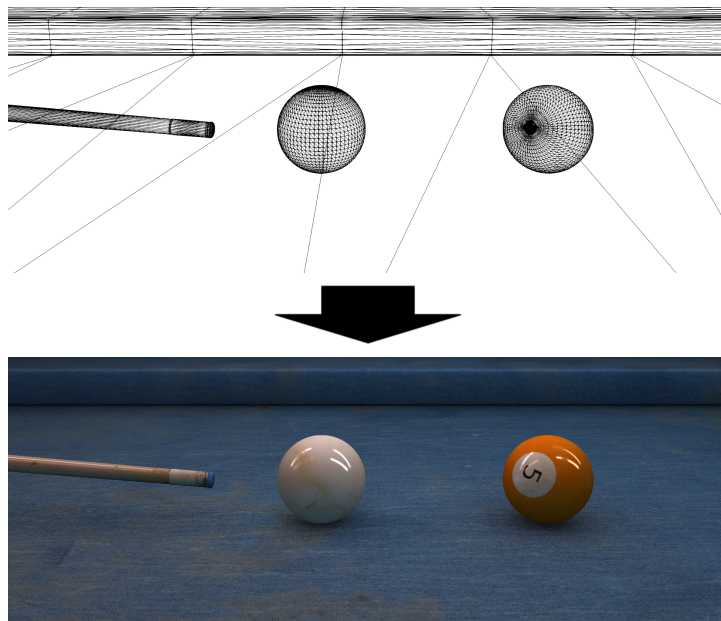
Rendering is a procedure of transforming 3D objects to 2D image suitable for showing on screen. In this thesis we discuss a field of real-time rendering solutions with emphasis on deferred rendering. We present a classic approach with forward rendering and two most typical deferred rendering approaches: *Deferred Shading* and *Deferred Lighting*. Next we develop our solution of deferred rendering with aid of *C#* and *HLSL* programming languages as well as *XNA* framework. We describe creation of necessary buffers, the procedure of geometry drawing, the procedure of lighting and the creation of geometry for the light volumes. Besides the classic light types, we develop our own solution of neon light, which offers interesting possibilities for lighting. At the end we present some possible improvements.

Key words: rendering, deferred rendering, deferred shading, deferred lighting, neon light, *C#*, *HLSL*, *XNA*.

Poglavje 1

Uvod

Upodabljanje je postopek pretvorbe 3D objektov v 2D sliko primerno za prikaz na zaslonu.



Slika 1.1: Rezultat upodabljanja¹.

¹Strogo tehnično gledano je tudi zgornji del slike (žičnati prikaz scene) rezultat upodabljanja.

Aplikacije, katerih končen rezultat je čim bolj natančna preslikava 3D scene v 2D sliko, običajno uporabijo najbolj napredne algoritme za preiskovanje, osvetljevanje in senčenje 3D scene. Tako zagotovijo karseda pristno preslikavo v 2D prostor. Seveda pa so takšni algoritmi večinoma tudi časovno potratni, saj lahko traja tudi več ur preden dobimo končno sliko. Takšen način upodabljanja se večinoma uporablja v filmski produkciji, arhitekturnih vizualizacijah, digitalni umetnosti in povsod drugod, kjer je pomembna predvsem kvaliteta končne slike.

Aplikacije, katerih ključen del je odzivnost sistema ter sproten prikaz informacij, si ne morejo privoščiti rešitev upodabljanja, kjer se končna slika generira več sekund, minut ali celo ur. Za takšne aplikacije je pomembno realno-časovno upodabljanje, kjer se za končno sliko porabi manj kot 50 milisekund, oziroma sposobnost prikazati več kot 20 slik na sekundo. Tipičen primer takšnih aplikacij so računalniške igre, kjer se uporabnik navadno ne zadovolji z manj kot 30 slikami na sekundo. Najbolj pogosta hitrost, ki jo uporabniki pričakujejo, pa je 60 slik na sekundo, kolikor je tudi osvežilni čas večine računalniških zaslonov. Hitro generiranje slik logično pomeni, da je nekje potrebno narediti kompromis. Lahko zmanjšamo velikost slike, vendar si z majhno sliko ponavadi ne moremo kaj dosti pomagati, saj izgubimo tudi veliko pomembnih detajlov. Takšna rešitev tudi sicer ni popularna, saj je najbolj naraven način prikaza, kjer velikost slike ustreza največji ločljivosti zaslona. Bolj sprejemljiva rešitev je zmanjšanje realizma slike. To lahko dosežemo z uporabo manj podrobnih 3D objektov, poenostavljeno osvetlitvijo in senčenjem ter raznimi triki za približno simuliranje realizma.

Od samega začetka realno-časovnih rešitev za upodabljanje je veljalo, da je osvetljevanje scene z več lučmi časovno zelo potratno. Tako se je uporabljalo le majhno število luči. Z napredkom tehnologije pa so postale mogoče tudi rešitve, ki omogočajo veliko večje število luči. V diplomskem delu tako obravnavamo področje realno-časovnih rešitev upodabljanja s poudarkom na

postopku odloženega upodabljanja.

1.1 Takojšnje upodabljanje

Takojšnje upodabljanje (angl. *Forward Rendering*) je klasičen pristop realno-časovnega upodabljanja, pri katerem se geometrija vsakega objekta osvetljuje z vsemi viri osvetlitve, ki prispevajo h končni osvetlitvi. Obstajata dva pristopa k osvetljevanju:

- Za vsako luč izrišemo geometrijo in jo osvetlimo, rezultate posameznih osvetlitev pa aditivno združimo v končno osvetlitev.
- Vsak kos geometrije izrišemo in osvetlimo z več lučmi hkrati. Pri tem prihranimo nekaj izrisovanja geometrije, vendar je en korak osvetljevanja toliko bolj zahteven kolikor več luči upoštevamo.

Ne glede na pristop, pa je časovna zahtevnost takojšnjega upodabljanja:

$$O(\text{geometrija} \cdot \text{število luči}), \quad (1.1)$$

zaradi česar smo zelo omejeni s številom luči, ki osvetljujejo posamezen objekt, kot tudi s samo kompleksnostjo geometrije objekta.

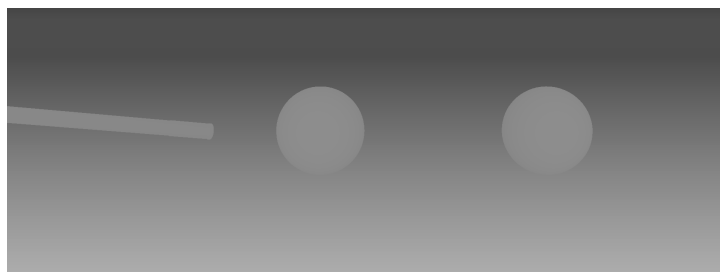
Dobra lastnost takojšnjega upodabljanja je širok nabor materialov in osvetlitvenih modelov, možnost upodabljanja prosojnih objektov ter uporaba strojnega glajenja robov.

Poglavje 2

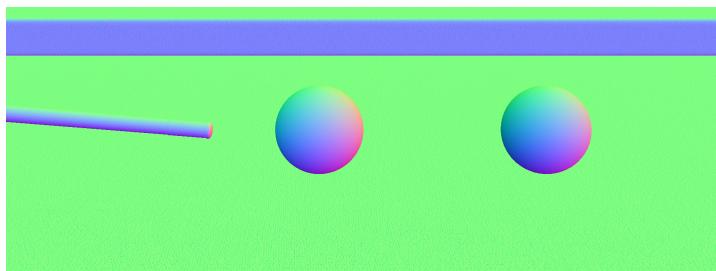
Odloženo upodabljanje

Odloženo upodabljanje je alternativa takojšnjemu upodabljanju. Glavna značilnost odloženega upodabljanja je razbitje postopka na dve stopnji.

Na prvi stopnji se v geometrični medpomnilnik (angl. *Geometry Buffer* ali krajše *G-Buffer*) zapiše le informacija o geometriji objekta ter lastnostih materiala. Informacijo o geometriji predstavljata lokacija in vektor normale objekta. Lastnosti materiala so odvisne od izbire modela osvetljevanja, ki ga bomo uporabili na naslednji stopnji. Običajno se shrani vsaj barvo objekta, intenziteto zrcalnega odboja in eksponent zrcalnega odboja. Po koncu prve stopnje imamo v geometričnem medpomnilniku le informacijo o vidnem delu scene.



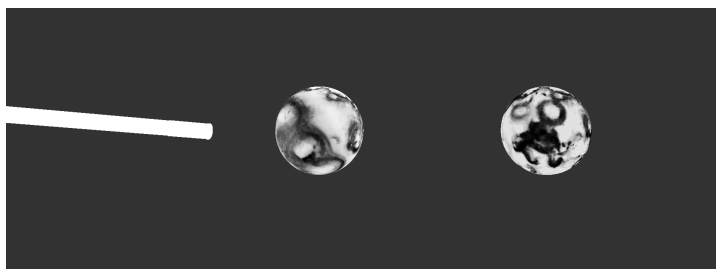
Slika 2.1: Geometrični medpomnilnik - oddaljenost.



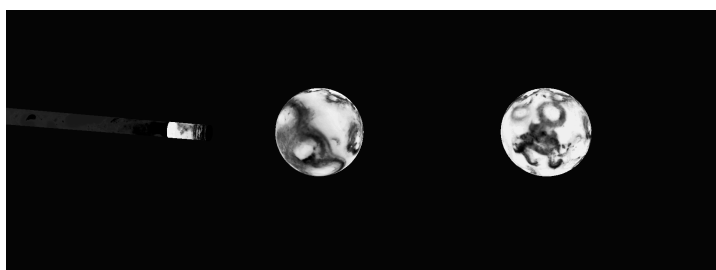
Slika 2.2: Geometrični medpomnilnik - vektor normale.



Slika 2.3: Geometrični medpomnilnik - barva objekta.

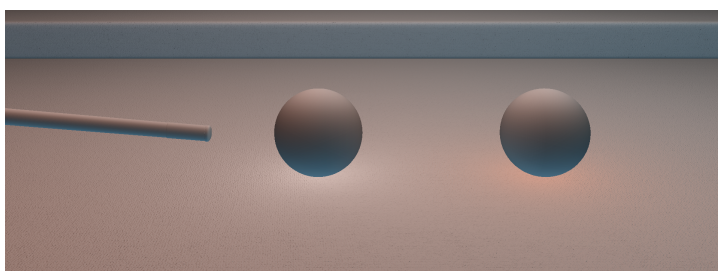


Slika 2.4: Geometrični medpomnilnik - intenziteta zrcalnega odboja.

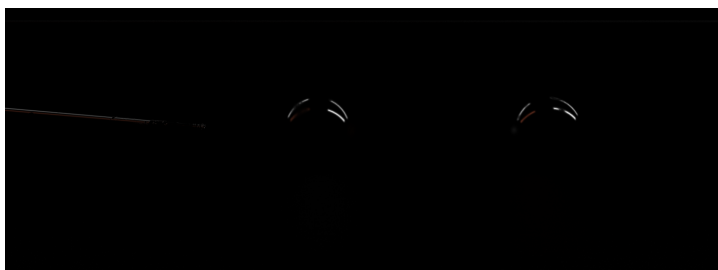


Slika 2.5: Geometrični medpomnilnik - eksponent zrcalnega odboja.

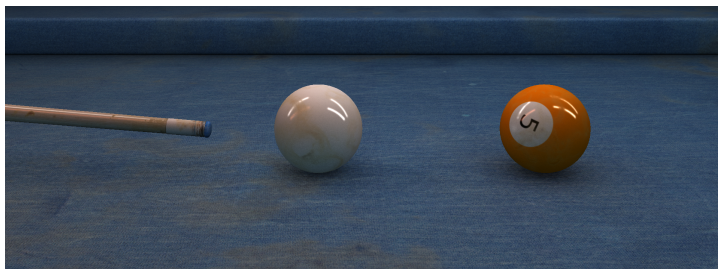
Na drugi stopnji se lotimo osvetljevanja scene z uporabo informacije iz geometričnega medpomnilnika. Osvetlitev lahko izrišemo kot končno sliko ali pa posamezne komponente shranimo v osvetlitveni medpomnilnik, s pomočjo katerega nato definiramo izgled končne slike. Izbira je odvisna od kompleksnosti osvetlitvenega postopka.



Slika 2.6: Osvetlitveni medpomnilnik - razpršeni odboji svetlobe.



Slika 2.7: Osvetlitveni medpomnilnik - zrcalni odboji svetlobe.



Slika 2.8: Končna slika.

Pri odloženem upodabljanju se za vsako luč osvetli vsako pokrito slikovno piko le enkrat, saj geometrični medpomnilnik hrani le najmanj oddaljeno geometrijo. Za razliko od takojšnjega upodabljanja, kjer se osvetljujejo tudi točke geometrije, ki kasneje niso vidne, če si delijo slikovno piko z neko bližjo točko. Z upoštevanjem vsega navedenega, časovna kompleksnost upodabljanja tako postane:

$$O(\text{geometrija} + \text{število luči}). \quad (2.1)$$

2.1 Prednosti odloženega upodabljanja

Dobra lastnost odloženega upodabljanja je zagotovo majhna časovna kompleksnost, zaradi katere si lahko privoščimo veliko večje število luči kot pri klasičnem takojšnjem upodabljanju. Prav tako si lahko privoščimo nekaj več geometrije, saj izris le-te ni vezan na osvetljevanje. Obstaja tudi veliko število postopkov za končno obdelavo slike, ki za delovanje potrebujejo prav informacije iz geometričnega medpomnilnika. Pri takojšnjem upodabljanju bi potrebovali dodaten izris geometrije, le da bi pridobili potrebne informacije. Zaradi hranjenja informacije o lokaciji geometrije, je odloženo upodabljanje primerno tudi za lepljenje senc (angl. *Shadow Mapping*) [1], ki je najpogosteje uporabljan postopek metanja senc.

2.2 Slabosti odloženega upodabljanja

Slaba lastnost odloženega upodabljanja je konstantna obremenitev pasovne širine grafičnega pomnilnika, kar je posledica uporabe geometričnega medpomnilnika. To je lahko moteče, še posebno kadar model osvetljevanja potrebuje veliko število parametrov, ki jih moramo shraniti v geometrični medpomnilnik. Poleg tega ne omogoča izrisa prosojnih objektov, saj vsaka slikovna pika geometričnega medpomnilnika lahko hrani le informacijo o najbližji vidni točki geometrije. To pa ne pomeni, da scena ne sme vsebovati prosojne geometrije. Le-to se lahko izriše po končanem odloženem upodabljanju z

uporabo takojšnjega upodabljanja. Vse slikovne pike geometričnega medpomnilnika vsebujejo isti tip parametrov za predstavitev materiala objekta, zato smo omejeni na majhno število osvetlitvenih modelov (običajno le en model). Strojne rešitve glajenja robov, kot je *MSAA* (angl. *Multisample Anti-Aliasing*) [2], niso smotrne, saj so le-te prostorsko zahtevne že pri takojšnjem upodabljanju. Pri odloženem upodabljanju pa je prostorska obremenitev še toliko večja zaradi velikosti geometričnega medpomnilnika. Hkrati pa za delovanje potrebujejo končno barvo geometrije v trenutku izrisovanja. To pri odloženem upodabljanju ni možno, saj dobimo končno barvo šele po končanem osvetljevanju. Vendar pa je bila v zadnjih letih razvita cela vrsta naprednih algoritmov, s katerimi lahko obdelamo končno sliko ter tako zgladimo robove.

2.3 Pristopi k odloženemu upodabljanju

Obstaja več pristopov k implementaciji odloženega upodabljanja, vendar ima vsak svoje dobre in slabe lastnosti. Tukaj bomo omenili le dva najbolj uporabljana.

2.3.1 Odloženo senčenje

Odloženo senčenje (angl. *Deferred Shading*) [3] uporablja osnovni postopek odloženega upodabljanja in je pogosto sinonim za odloženo upodabljanje. Za odloženo senčenje tako veljajo enake prednosti in slabosti kot pri osnovnem postopku. Za razliko od osnovnega postopka odloženega upodabljanja, odloženo senčenje nekoliko bolj definira postopek osvetljevanja ter možnosti za zmanjšanje časovne zahtevnosti posamezne luči.

Osvetljevanja bi se lahko lotili tako, da bi za vsak vir svetlobe osvetlili vse slikovne pike, vendar je to časovno potratno. Tako bi namreč isto časa namenili slikovnim pikam, ki so osvetljene, kot tudi tistim, ki izgledajo neosvetljene zaradi omejene natančnosti medpomnilnika. Zato se za vsako luč

generira geometrijo, ki se kar najbolj prilega tipu luči ter tako pokrije kar seda majhno število slikovnih pik. S tem dosežemo, da je časovna zahtevnost luči, ki pokrije veliko število slikovnih pik, približno enaka kot pri velikem številu majhnih luči, ki skupno pokrijejo enako število slikovnih pik. Zato je odloženo upodabljanje uporabno predvsem kadar imamo veliko število majhnih lokalnih luči. Geometrijo luči lahko izrišemo kot zaslonsko poravnan pravokotnik (angl. *Screen-Aligned Quad*) ali kot konveksen volumen luči (angl. *Convex Light Volume* ali *Convex Light Hull*), ki ustreza tipu luči.

Zaslonsko poravnan pravokotnik

Zaslonsko poravnan pravokotnik predstavlja grobo aproksimacijo projekcije luči na zaslonsko ravnino, pri čemer zajame tudi precejšen delež nepotrebnih slikovnih pik. Dobra lastnost pa je, da ima majhno geometrično zahtevnost, saj ga sestavljata le dva trikotnika.

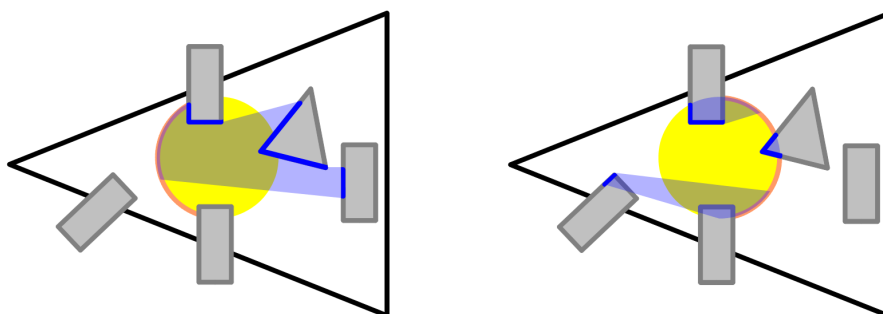
Konveksen volumen luči

Pristop s konveksnim volumenom luči je bolj običajen, saj se za vsak tip luči generira drugačna geometrija. Le-ta po projekciji na zaslonsko ravnino pokrije le majhen delež nepotrebnih slikovnih pik.

Slaba lastnost je večja geometrična zahtevnost, saj z večjo natančnostjo prilagajanja volumnu luči narašča tudi število trikotnikov. Vendar pa se da že z dokaj majhno kompleksnostjo dobiti precej boljšo pokritost slikovnih pik v primerjavi z zaslonsko poravnanim pravokotnikom. S konveksnostjo geometrije zagotovimo, da se pokrita slikovna pika osvetli le enkrat, čeprav jo pokrivata dva trikotnika geometrije, saj lahko določimo ali želimo izrisati notranje ali zunanje trikotnike. Prav tako se lahko določi, katere pokrite pike ne bomo osvetlili, če se lokacija iz geometričnega medpomnilnika nahaja izven volumna luči. To se lahko stori z dvema različnima pristopoma:

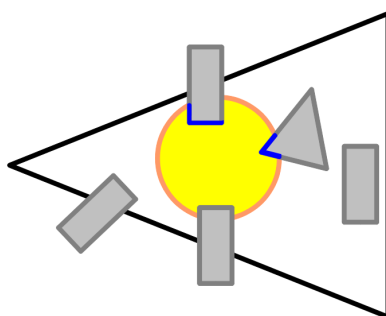
- Pri prvem se primerja le oddaljenost lokacije in mejne ploskve volumna luči. Določimo lahko le ali se lokacija nahaja pred ali za mejno

ploskvijo. To pomeni, da je s takšnim pristopom mogoče odpraviti le del točk, ki jih ni potrebno osvetliti. Vendar pa enostavnost postopka omogoča pakiranje luči, saj ni potrebe po ločenem obravnavanju posamezne luči.



Slika 2.9: Osvetljevanje s prednjo oziroma zadnjo ploskvijo volumna luči.

- Drugi pristop konstruira masko, s katero točno določi le tiste slikovne pike, ki jih je potrebno osvetliti. Tako se časovno zahtevni senčilniki izvajajo na karseda majhnem številu slikovnih pik. Vendar pa ima ta pristop tudi slabo plat, saj zaradi postopka konstrukcije maske ni mogoče pakiranje luči. To pomeni, da je potrebno narediti masko za vsako luč posebej ter vsako luč posebej tudi izrisati.



Slika 2.10: Osvetljevanje s konstrukcijo maske iz volumna luči.

2.3.2 Odloženo osvetljevanje

Odloženo osvetljevanje (angl. *Deferred Lighting* [4] ali *Light Pre-Pass* [5])¹ poskuša odpraviti slabosti odloženega senčenja. Postopek je razdeljen na tri stopnje:

1. V geometrični medpomnilnik shranimo le informacijo o geometriji objekta, torej le lokacijo in vektor normale².
2. Osvetljevanje scene in generiranje osvetlitvenega medpomnilnika, v katerega shranimo barvo razpršenega odboja svetlobe ter vrednost zrcalnega odboja svetlobe.
3. Postopek takojšnjega upodabljanja, le da za osvetljevanje uporabimo informacijo iz osvetlitvenega medpomnilnika.

S tem pristopom se zmanjša velikost geometričnega medpomnilnika ter posledično obremenitev pasovne širine. Način shranjevanja vrednosti zrcalnega odboja svetlobe v kombinaciji s tretjim korakom omogoči potencialno večji nabor materialov. Vendar smo pri materialih še vedno omejeni na majhno število modelov osvetljevanja, če ne želimo povečati prostorske zahtevnosti.

Zaradi varčevanja pri velikosti geometričnega in osvetlitvenega medpomnilnika se pojavijo nove slabosti, ki so lahko bolj ali manj moteče, odvisno od situacije. Ena takšnih lastnosti je barva zrcalnega odboja svetlobe, ki je v osnovi vedno bela, ne glede na barvo luči. Za to sicer obstaja rešitev, saj je barvo mogoče rekonstruirati iz barve razpršenega odboja. Vendar pa nastane problem, če se v isti točki prekriva več luči, kar povzroči, da je rekonstruirana barva mešanica barv teh luči. Kot slabo lastnost se lahko smatra tudi potreba po dvakratnem izrisovanju geometrije, še posebno če želimo izrisati veliko količino le-te.

¹V bistvu gre za dva ločena postopka, ki pa se zaradi velike podobnosti pogosto omenjata pod skupnim imenom.

²V primeru postopka *Deferred Lighting*, se shrani še eksponent zrcalnega odboja (običajno v *Alpha* kanal texture z vektorjem normale).

Poglavje 3

Razvojno okolje

Razvoj je potekal v okolju *Visual C# 2010 Express* [6], ki je okrnjena verzija okolja *Visual Studio* [7]. Uporabili smo ogrodje *.NET* [8], namestili pa smo si tudi ogrodje *XNA 4.0* [9], ki ponuja strukture in metode za lažje delo z vmesnikom *DirectX 9* [10].

Pisanje osnovne logike je potekalo v programskem jeziku *C#* [11]. Senčilnike, ki predstavljajo navodila grafičnemu procesorju, smo spisali v programskem jeziku *HLSL* [12]. Ogrodje *XNA* podpira senčilnike največ tretje generacije (angl. *Shader Model 3*) [13], zato je bil razvoj rešitve temu ustrezno prilagojen.

Tridimenzionalne modele, potrebne za testiranje in razvoj rešitve, smo izdelali s programskim orodjem *Blender* [14], texture zanje pa smo kreirali s pomočjo programskega orodja *GIMP* [15] ter tekstur, dostopnih na *CGTextures* [16].

Poglavje 4

Razvoj rešitve

4.1 Izdelava geometričnega medpomnilnika

Geometrični medpomnilnik realiziramo kot skupek tekstur tipa *RenderTarget2D* [17][18], na katere izrisujemo posamezne lastnosti geometrije. Texture morajo biti enake bitne globine, če želimo uporabiti lastnost *MRT* (angl. *Multiple Render Targets*) [19], ki nam omogoči pisanje v več tekstur hkrati. Uporaba lastnosti *MRT* za realizacijo odloženega upodabljanja sicer ni potrebna, vendar pa bi brez nje potrebovali toliko izrisov geometrije kolikor imamo tekstur, ki sestavljajo geometrični medpomnilnik. Z uporabo lastnosti *MRT* je tako potreben le en izris geometrije.

Izbira, katere informacije in v kakšni natančnosti bo vseboval geometrični medpomnilnik, določa kako kvalitetno bomo lahko osvetlili geometrijo ter koliko različnih materialov bomo lahko upodobili. Za geometrični del potrebujemo lokacijo in vektor normale. Pri izbiri lastnosti materiala je potrebno upoštevati tudi model osvetljevanja, ki ga bomo uporabili v fazi osvetljevanja. Odločili smo se uporabiti *Blinn-Phong*-ov model [20], kar pomeni, da poleg barve objekta potrebujemo še intenziteto in eksponent zrcalnega odboja. Intenziteto zrcalnega odboja smo se odločili hraniti kot barvo, s čimer lahko dosežemo obarvanost zrcalnih odbojev. Dodali pa smo tudi moč izžarevanja

(angl. *emissive*), s katero lahko dosežemo, da objekt izžareva lastno barvo tudi kadar ni osvetljen.

Pomembno je tudi, da je medpomnilnik karseda majhen, saj se tako zmanjša obremenitev pasovne širine grafičnega pomnilnika. Za hranjenje barve objekta in moči izžarevanja zadošča tekstura s formatom *R8G8B8A8* [21], tako da v *RGB* kanale shranimo barvo objekta, v *Alpha* kanal pa shranimo 8-bitno moč izžarevanja. Enak format uporabimo tudi za barvo zrcalnega odboja (*RGB* kanali) in eksponent zrcalnega odboja (*Alpha* kanal). Za vektor normale želimo nekoliko večjo natančnost kot jo ponuja osem bitov na kanal, saj se sicer zdi, da površina objekta ni gladka. Prav tako želimo čim večjo natančnost pri shranjevanju lokacije. Čeprav bi bila 32-bitna natančnost za vsako od *XYZ* komponent vektorjev normale in lokacije idealna, je to občutno preveč, saj bi potrebovali šest 32-bitnih tekstur samo za hranjenje vektorjev normale in lokacije. Zato naredimo kompromis in izberemo format *R10G10B10A2* [21], ki nam ponuja deset bitov za vsako od *XYZ* komponent vektorja normale. Poleg tega pa ponuja še 2-bitni *Alpha* kanal, v katerega lahko spravimo do štiri kontrolne vrednosti, ki so prav tako uporabne, kot bomo pojasnili v nadaljevanju. Za lokacijo pa je tudi 10-bitna natančnost premalo, zato bomo hranili le 32-bitno oddaljenost objekta, iz katere lahko rekonstruiramo pravo lokacijo. Za ta namen izberemo format *Single* [21]. V nadaljevanju bomo texture geometričnega medpomnilnika imenovali kot G_1 , G_2 , G_3 in G_4 .

G_1 :	Barva objekta (<i>RGB</i>)	Moč izžarevanja
G_2 :	Barva zrcalnega odboja (<i>RGB</i>)	Eksponent zrcalnega odboja
G_3 :	Vektor normale (<i>XYZ</i>)	Kontrolne vrednosti
G_4 :	Oddaljenost	

Tabela 4.1: Shematski prikaz geometričnega medpomnilnika.

4.2 Izris geometrije in materialnih lastnosti

V enem koraku želimo v geometrični medpomnilnik shraniti vse potrebne informacije o geometriji in materialu objekta, zato potrebujemo model senčilnikov tretje generacije, ki podpira lastnost *MRT*.

4.2.1 Priprava medpomnilnika

Preden se lotimo izrisovanja objektov v geometrični medpomnilnik, je le-tega potrebno napolniti s privzetimi vrednostmi. To storimo tako, da izrišemo celozaslonski pravokotnik (angl. *Fullscreen Quad*) z upoštevanjem zamika za pol slikovne pike (angl. *Half-Pixel Offset*) [22] [23] in v senčilniku slikovnih pik (angl. *Pixel Shader*) nastavimo vrednosti. Vrednost teksture G_4 nastavimo na 1, saj je to največja oddaljenost, ki jo še zajamemo v vidno polje. Vrednost ostalih tekstur nastavimo na 0.

4.2.2 Izris objektov

Senčenje oglišč

V senčilniku oglišč (angl. *Vertex Shader*) naredimo preslikavo lokacije iz 3D na 2D. Na izhod podamo še vektor normale ter ZW^1 komponenti preslikane lokacije. Komponenti ZW potrebujemo v senčilniku slikovnih pik, vendar tam ne moremo dostopati do preslikane lokacije. Na izhod posredujemo še vektorja tangente in binormale objekta, če se v senčilniku slikovnih pik uporablja tudi tehnika "*normal mapping*" [24], s katero spreminjamo vektor normale. Seveda pa v primeru, da uporabljamo teksturiranje objektov, ne smejo manjkati koordinate vzorca teksture.

¹Tukaj in v nadaljevanju okrajšujemo zapis posameznih kombinacij komponent 4D vektorja (X, Y, Z, W) .

```

output.Position = mul(input.Position , ViewProjection);
output.Normal = input.Normal;
output.Tangent = input.Tangent;
output.Binormal = input.Binormal;
output.TexCoord = input.TexCoord;
output.Depth.xy = output.Position.zw;

```

Izsek kode 4.1: Primer senčilnika oglišč.

Senčenje slikovnih pik

V senčilniku slikovnih pik nastavimo vrednosti za vse teksture geometričnega medpomnilnika. Vrednosti, ki jih lahko shranimo, so odvisne of formata posamezne teksture. Za formata *R8G8B8A8* in *R10G10B10A2* so možne vrednosti enakomerno razporejene na intervalu $[0, 1]$, medtem ko se za format *Single* uporablja 32-bitni format plavajoče vejice. Teksturi G_1 in G_2 sta brez posebnosti in v njune komponente zapišemo vrednosti kot jih kaže tabela 4.1. V *RGB* kanale teksture G_3 shranimo vektor normale objekta. Pred tem je vektor potrebno normalizirati ter narediti preslikavo iz definicijskega območja $[-1, 1]$, ki mu pripada vsaka od *XYZ* komponent vektorja, na definicijsko območje $[0, 1]$, ki ga lahko hrani tekstura. Normalizacija vektorja normale pred preslikavo je potrebna zaradi linearne interpolacije vrednosti med posameznimi oglišči, zaradi česar se spreminja tudi velikost vektorja. Na tem mestu naj omenimo še, da je pred samo preslikavo vektorja normale potrebno upoštevati tudi vektorja tangente in binormale, če želimo vplivati na smer vektorja. *Alpha* kanal teksture G_3 služi kot kontrolni del v nadaljevanju ter nam predstavlja prisotnost oziroma odsotnost geometrije. Vanj lahko shranimo štiri vrednosti: 0 , $\frac{1}{3}$, $\frac{2}{3}$ in 1 . Vrednost 0 nam predstavlja odsotnost geometrije, ostale vrednosti pa prisotnost geometrije. Pri tem velja omeniti, da je dovolj le ena od teh treh vrednosti, tako da imamo na voljo še dve, če bi želeli geometrični medpomnilnik razširiti s še kakšnim kontrolnim parametrom. Tekstura G_4 je v celoti namenjena oddaljenosti, ki jo definiramo kot količnik prej omenjenih *ZW* komponent preslikane lokacije.

```

float3 sample = tex2D(NormalSampler, input.TexCoord).rgb;
float3 normalBump = sample * 2 - 1;
float3 normal = normalize(input.Normal);
float3 tangent = normalize(input.Tangent) * normalBump.x;
float3 binormal = normalize(input.Binormal) * normalBump.y;
normal = normalize(normal + tangent + binormal);
output.Normal = float4(normal * 0.5 + 0.5, 1);

output.Depth = input.Depth.x / input.Depth.y;

```

Izsek kode 4.2: Preslikava vektorja normale in izračun oddaljenosti.

4.3 Izdelava osvetlitvenega medpomnilnika

Enako kot geometrični medpomnilnik, je tudi osvetlitveni medpomnilnik sestavljen iz več tekstur. V osvetlitvenem medpomnilniku želimo hraniti barvi razpršenih in zrcalnih odbojev svetlobe. Potrebujemo torej dve teksturi formata *R8G8B8A8*. Odločimo se lahko tudi za format *R10G10B10A2* ali *HdrBlendable* [21] [25], če potrebujemo večjo natančnost. Ta dva formata še podpirata aditivno zapisovanje. Potrebujemo pa le *RGB* kanale za shranjevanje barve, tako da le 2-bitni *Alpha* kanal formata *R10G10B10A2* ni problematičen. V nadaljevanju bomo teksturi osvetlitvenega medpomnilnika imenovali kot L_1 in L_2 .

L_1 :	Barva razpršenega odboja (<i>RGB</i>)	/
L_2 :	Barva zrcalnega odboja (<i>RGB</i>)	/

Tabela 4.2: Shematski prikaz osvetlitvenega medpomnilnika.

4.4 Izdelava geometrije volumnov luči

Osvetljevanja smo se lotili z uporabo volumnov luči, zato je bilo potrebno izdelati geometrijo za vsak tip luči, ki naj bi ga naš upodabljevalnik podpiral. Ob tem sta lokacija in velikost geometrije pomembna faktorja v nadaljevanju, ko je geometrijo potrebno prilagoditi posamezni instanci luči.

4.4.1 Ambientalna luč

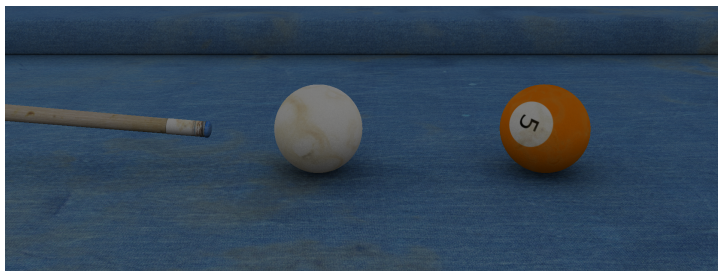
Ambientalna luč predstavlja povprečno osvetlitev celotne scene, zato je najbolj primerna geometrija takšna, ki pokrije celotno površino zaslona in smo jo realizirali kot celozaslonski pravokotnik.

4.4.2 Usmerjena luč

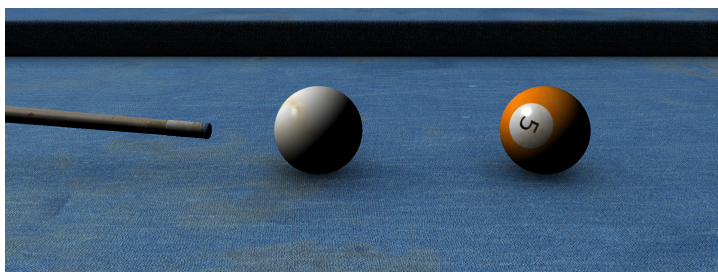
Usmerjena luč ima izvor svetlobe neskončno daleč, zato so žarki svetlobe vzporedni. Iz istega razloga kot pri ambientalni luči, smo tudi za usmerjeno luč izbrali celozaslonski pravokotnik.

4.4.3 Točkasta luč

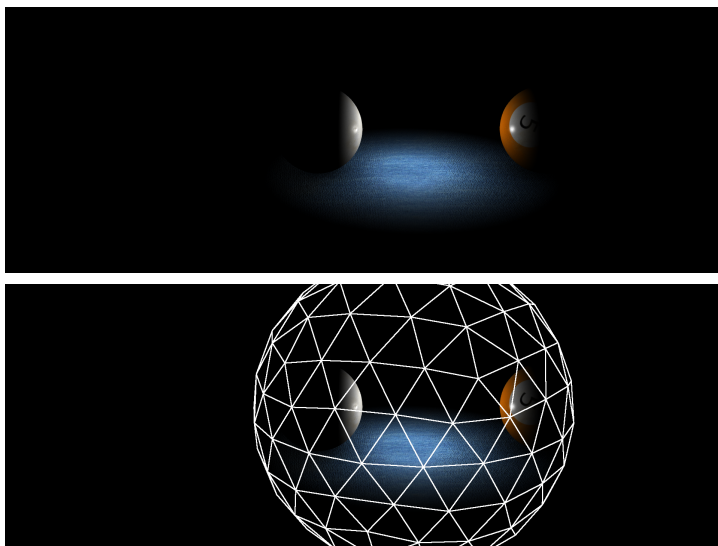
Točkasta luč ima izvor svetlobe nekje v prostoru in sveti v vse smeri z enako intenziteto, ki pada s kvadratom razdalje od izvora. Za geometrijo smo izbrali kroglo ter jo implementirali kot razdeljen ikozaeder (angl. *subdivided icosahedron*), s katerim lahko poljubno natančno izrišemo približek krogli. Pri tem se je potrebno zavedati, da z večjo natančnostjo hkrati generiramo tudi večjo količino geometrije, kar zna upočasniti proces izrisovanja. Z majhno natančnostjo pa pri izrisovanju pokrijemo večje število slikovnih pik, ki so izven polmera krogle. Velikost ikozaedra smo nastavili tako, da zajame celoten volumen enotske krogle centrirane v izhodišču koordinatnega sistema.



Slika 4.1: Primer osvetlitve z ambientalno lučjo.



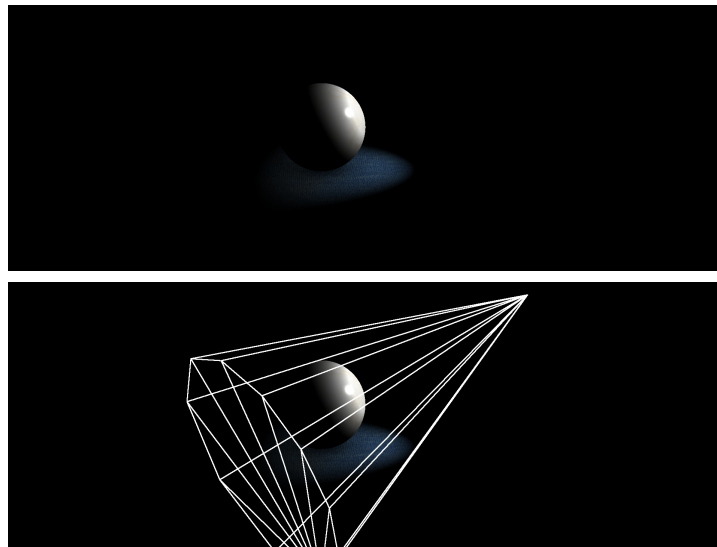
Slika 4.2: Primer osvetlitve z usmerjeno lučjo.



Slika 4.3: Primer osvetlitve in uporabe volumna točkaste luči.

4.4.4 Reflektorska luč

Reflektorska luč je podobna točkasti luči, le da je usmerjena in ima omejen kot. Lahko bi zopet izbrali kroglo, vendar bi s tem pokrili večino slikovnih pik, ki ne bodo osvetljene. Zato je bolj primerna oblika stožca. Geometrijo smo implementirali tako, da je vrh stožca v izhodišču koordinatnega sistema. Osnovna ploskev je predstavljena kot pravilni mnogokotnik, očrtan enotski krožnici s središčem v točki $(0, -1, 0)$ ter leži vzporedno z XZ ravnino. Tako kot pri točkasti luči smo tudi tukaj prilagodili implementacijo tako, da omogoča poljubno nastavljanje natančnosti osnovne ploskve.

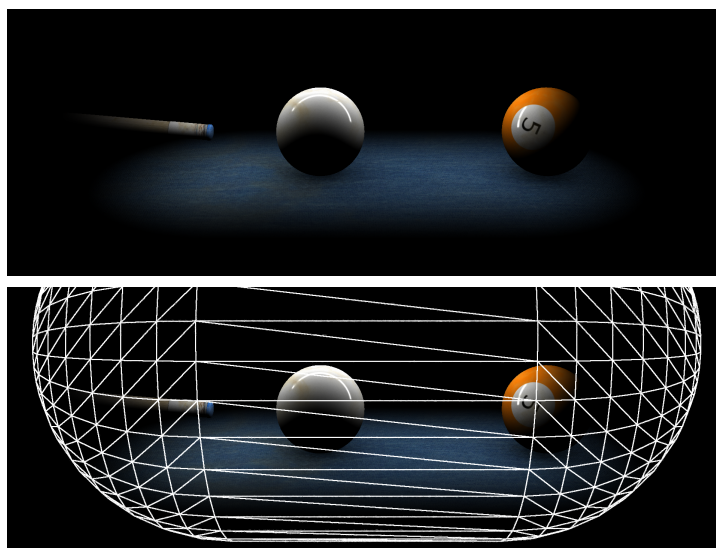


Slika 4.4: Primer osvetlitve in uporabe volumna reflektorske luči.

4.4.5 Neonska luč

Neonska luč oddaja svetlobo iz vsake točke na daljici v vse smeri. Za geometrijo smo izbrali obliko kapsule. Implementirali smo jo tako, da smo enotsko kroglo centrirano v izhodišču koordinatnega sistema prerezali z XZ ravnino. Obe polkrogli smo nato razmaknili za eno enoto po Y -osi ter povezali njuni

osnovni ploskvi. Pri tem je potrebno poudariti, da krogle nismo realizirali na enak način kot pri točkasti luči, temveč smo oglišča razporedili na geografski način poldnevnikov in vzporednikov. Takšen način nam omogoča enostavnejše nastavljanje natančnosti.



Slika 4.5: Primer osvetlitve in uporabe volumna neonske luči.

4.5 Izris osvetlitve

4.5.1 Priprava medpomnilnika

Tako kot pri geometričnem medpomnilniku, je tudi osvetlitveni medpomnilnik pred začetkom postopka osvetljevanja potrebno nastaviti na privzete vrednosti. Ta del postopka smo združili z osvetljevanjem z ambientalno lučjo tako, da teksturo L_1 nastavimo na barvo ambientalne svetlobe. Vrednost slikovnih pik texture L_2 pa nastavimo na 0, saj ambientalna svetloba ne prispeva zrcalnih odbojev.

4.5.2 Priprava senčilnikov luči

Za postopek osvetljevanja z ostalimi viri svetlobe potrebujemo določene informacije. Potrebujemo večino informacije iz geometričnega medpomnilnika, in sicer vse teksture razen G_1 . Pred postopkom osvetljevanja moramo zato senčilniku podati kazalce do ustreznih tekstur. Zakaž ne potrebujemo teksture G_1 , bomo pojasnili v nadaljevanju. Poleg tekstur posredujemo tudi matriko *ViewProjectionInverse*. Ta predstavlja inverz matrike, s katero smo preslikali 3D lokacijo v prostoru v 2D lokacijo na zaslonu. Matriko potrebujemo za rekonstrukcijo 3D lokacije. Za pravilen izračun zrcalnega odboja svetlobe potrebujemo tudi lokacijo kamere. Vsem tipom luči smo dodali tudi parameter *hasSpecular* za nadzor prikaza zrcalnega odboja svetlobe, saj si v določenih primerih želimo le osvetlitve z razpršenimi odboji svetlobe. Ostali parametri so odvisni od tipa luči.

Ne glede na tip luči v senčilniku oglišč izračunamo 2D lokacijo na zaslonu. Ta je potrebna tako zaradi same projekcije geometrije na zaslon, kot tudi za določitev slikovne pike, ki jo osvetlimo v senčilniku slikovnih pik. Prav tako pri vseh tipih luči, senčilnik slikovnih pik vsebuje zaznavanje geometrije, rekonstrukcijo 3D lokacije in preslikavo XYZ komponent vektorja normale iz definicijskega območja texture $[0, 1]$ na definicijsko območje $[-1, 1]$. Zaznavanje geometrije naredimo na preprost način, saj smo to informacijo shranili v kontrolne vrednosti geometričnega medpomnilnika (*Alpha* kanal texture G_3). V primeru, da ugotovimo odsotnost geometrije, pokličemo vgrajeno funkcijo *clip()* [26]. S tem prekinemo izvajanje senčilnika oziroma poskrbimo, da bo izvajanje postopka v nadaljevanju pohitreno in se bo rezultat po koncu postopka zavrzel. Kot smo že omenili, za osvetljevanje uporabljamo *Blinn-Phong*-ov model osvetljevanja. Sama izbira modela osvetljevanja na bistvo delovanja senčilnika slikovnih pik ne vpliva. Zato končno enačbo v nadaljevanju izpuščamo iz opisov za posamezne luči. Omenimo naj le, da iz enačbe izluščimo faktorja za razpršene in zrcalne odboje ter ju zapišemo na ustrezni teksturi L_1 in L_2 .

Rekonstrukcija 3D lokacije

Preden se lotimo rekonstrukcije 3D lokacije, iz geometričnega medpomnilnika preberemo shranjeno oddaljenost in jo primerjamo z oddaljenostjo geometrije volumna luči. Izvajanje senčilnika prekinemo, če ugotovimo, da je shranjena vrednost večja od oddaljenosti volumna. Pri tem naj omenimo, da smo se pri izrisu geometrije volumna odločili izrisovati le notranje oziroma bolj oddaljene trikotnike. Tako smo se izognili pretiranemu ugotavljanju ali se kamera nahaja znotraj volumna luči. Zaradi takšnega pristopa bi lahko nastal problem le kadar se volumen luči seka z najbolj oddaljeno ravnino kamere, saj bi tako nastalo področje brez osvetlitve. Vendar pa se da takšnim primerom izogniti z nekoliko razmisleka pri določanju velikosti volumnov luči kot tudi dosegom kamere.

Rekonstrukcija 3D lokacije poteka tako, da iz preslikane lokacije geometrije luči izračunamo XY koordinati lokacije v prostoru rezanja (angl. *Clip Space*). Nato izračunamo UV koordinati slikovne pike, ki ju uporabimo za dostop do shranjene oddaljenosti v teksturi G_4 . Zatem konstruiramo vektor $csPosition$, ki predstavlja lokacijo v prostoru rezanja ter ga pomnožimo z matriko $ViewProjectionInverse$. Dobimo vektor $wsPosition$, iz katerega rekonstruiramo 3D lokacijo.

```
float4 position = input.ViewPosition;
float2 csXY = position.xy / position.w;
float u = (csXY.x + 1 + PixelSize.x) * 0.5;
float v = (1 - csXY.y + PixelSize.y) * 0.5;
float2 texCoord = float2(u, v);
float depth = tex2D(DepthSampler, texCoord);
float4 csPosition = float4(csXY, depth, 1);
float4 wsPosition = mul(csPosition, ViewProjectionInverse);
float3 wPosition = wsPosition.xyz / wsPosition.w;
```

Izsek kode 4.3: Rekonstrukcija 3D lokacije.

Rekonstrukcija vektorja normale

Vektor normale rekonstruiramo kot kaže izsek kode 4.4. Pri tem velja omeniti, da je rekonstruiran vektor potrebno normalizirati, saj je pri shranjevanju v teksturo najverjetneje prišlo do zaokrožitve posameznih komponent zaradi omejene natančnosti.

```
float3 normal = tex2D(NormalSampler, texCoord).xyz;  
normal = normalize(normal * 2 - 1);
```

Izsek kode 4.4: Rekonstrukcija vektorja normale.

4.5.3 Razmnoževanje geometrije luči

Senčilnike za izris geometrije točkaste, reflektorske in neonske luči smo dodatno prilagodili tako, da podpirajo strojno razmnoževanje geometrije (angl. *Geometry Instancing*) [27]. S tem smo pohitrili postopek izrisovanja večih luči istega tipa. Razmnoževanje geometrije poteka tako, da grafični kartici posredujemo dva toka podatkov. Elementi na prvem toku predstavljajo oglišča geometrije. Elementi na drugem toku pa predstavljajo ustrezne parametre, potrebne za transformacijo posamezne instance. Grafični procesor nato poskrbi, da se vsak element prvega toka združi z vsakim elementom drugega toka.

Število in tip parametrov, potrebnih za transformacijo posamezne instance, sta odvisna od tipa luči, zato je tudi posredovanje parametrov senčilniku različno. V vseh primerih pa smo poskusili parametre zakodirati tako, da smo velikost elementa obdržali karseda majhno ter jih v senčilniku pred uporabo odkodirali.

4.5.4 Usmerjena luč

Za realizacijo usmerjene luči potrebujemo barvo svetlobe in vektor smeri, iz katere prihaja svetloba. Senčilnik smo implementirali tako, da podpira do tri usmerjene luči, katerih parametre nastavimo kot konstante senčilnika, rezultate posameznih osvetlitev pa seštejemo. Tako smo se izognili razmnoževanju geometrije oziroma večkratnemu izrisovanju v primeru, da uporabimo več kot le eno usmerjeno luč.

4.5.5 Točkasta luč

Točkasta luč je najbolj enostavna od vseh luči za razmnoževanje geometrije, saj zaradi okrogle oblike ne potrebuje parametra za rotacijo. Potrebni parametri so lokacija centra krogle, polmer krogle ter barva in intenziteta svetlobe. Parametre smo zakodirali v dva vektorja s štirimi komponentami:

$$\begin{aligned} V_1 &= (\text{lokacija}_{xyz}, \text{polmer}) \\ V_2 &= (\text{barva}_{rgb} \cdot \text{intenziteta}, \text{hasSpecular}) \end{aligned} \quad (4.1)$$

pri čemer ima parameter *hasSpecular* vrednost 1, če želimo zrcalni odboj, sicer ima vrednost 0.

Senčenje oglišč

Senčilnik oglišč uporabi parameter o lokaciji in polmeru za skaliranje in premik oglišča pred postopkom projekcije na zaslon.

```
float3 translation = input.PositionRadius.xyz;
float radius = input.PositionRadius.w;
float4 position = input.Position;
position = float4(position.xyz * radius + translation, 1);
```

Izsek kode 4.5: Skaliranje in premik oglišča.

Preostali parametri se posredujejo senčilniku slikovnih pik kjer se uporabijo za izračun osvetlitve.

4.5.6 Reflektorska luč

Parametre reflektorske luči predstavljajo lokacija vrha stožca, vektor usmerjenosti luči, polmer osnovne ploskve stožca, višina stožca ter barva in intenziteta svetlobe. Vse parametre, z izjemo parametra *hasSpecular*, lahko zakodiramo v 4x4 matriko:

$$M = \begin{bmatrix} R_i \cdot \text{polmer} & , & \text{barva}_r \cdot \text{intenziteta} \\ R_j \cdot \text{višina} & , & \text{barva}_g \cdot \text{intenziteta} \\ R_k \cdot \text{polmer} & , & \text{barva}_b \cdot \text{intenziteta} \\ \text{lokacija}_{xyz} & , & D \end{bmatrix} \quad (4.2)$$

kjer vektorji R_i , R_j in R_k predstavljajo ortonormirano bazo rotacijske matrike in vektor R_j kaže v nasprotni smeri vektorja usmerjenosti luči. Parameter D pa smo definirali kot:

$$D = \frac{\text{višina}}{\sqrt{\text{polmer}^2 + \text{višina}^2}}, \quad (4.3)$$

in predstavlja kosinus največjega odklona od usmerjenosti luči, znotraj katerega še osvetljujemo.

Senčenje oglišč

Z množenjem rotacijskega dela matrike M in oglišča, naredimo rotacijo in skaliranje v enem koraku. Nato naredimo le še premik na dano lokacijo. Namesto normaliziranega vektorja usmerjenosti posredujemo senčilniku slikovnih pik kar vektor $-y$ (definiran v izseku kode 4.6). Omeniti velja, da vektorja ne normaliziramo, ker v senčilniku slikovnih pik potrebujemo tudi informacijo o višini stožca. Barvo skalirano z intenziteto in parameter D zakodiramo v vektor s štirimi komponentami. Senčilniku slikovnih pik posredujemo še vektor t (definiran v izseku kode 4.7) in parameter *hasSpecular*.


```
float3 x = input.DataRow0.xyz;
float3 y = input.DataRow1.xyz;
float3 z = input.DataRow2.xyz;
float3 t = input.DataRow3.xyz;
float3x3 rotation = {x, y, z};
float3 pos = mul(input.Position, rotation);
float4 position = float4(pos + t, 1);
```

Izsek kode 4.6: Rotacija, skaliranje in premik oglišča.

```
float r = input.DataRow0.w;
float g = input.DataRow1.w;
float b = input.DataRow2.w;
float a = input.DataRow3.w;
float4 v = float4(r, g, b, a);
```

Izsek kode 4.7: Barva, intenziteta in kosinus največjega odklona.

4.5.7 Neonska luč

Za predstavitev daljice v prostoru smo uporabili krajišči daljice. Ostali parametri so polmer polkrožnih zaključkov ter barva in intenziteta svetlobe. Dodali pa smo tudi nekaj parametrov, ki razširijo delovanje neonske luči za posebne primere osvetlitve. Ti parametri kontrolirajo intenziteto zrcalnega odboja, pretvorijo geometrijo iz kapsule v valj ter dodajo masko zastiranja svetlobe. Večino parametrov zapakiramo v eno 4x4 matriko:

$$M = \begin{bmatrix} R_i \cdot polmer & , & barva_r \cdot intenziteta \\ R_j \cdot H & , & barva_g \cdot intenziteta \\ R_k \cdot I & , & barva_b \cdot intenziteta \\ S & , & endCaps \end{bmatrix}, \quad (4.4)$$

kjer vektorji R_i , R_j in R_k predstavljajo ortonormirano bazo rotacijske matrike, pri čemer R_j kaže v smeri premice, na kateri leži daljica. Vektorja R_i

in R_k določata rotacijo okoli R_j , s čimer nadzorujemo usmerjenost maske za zastiranje svetlobe. Parameter S predstavlja središčno točko daljice:

$$S = \frac{A + B}{2}, \quad (4.5)$$

kjer sta A in B krajišči daljice. Parameter H predstavlja polovico dolžine daljice:

$$H = ||S - A||. \quad (4.6)$$

Parameter I je intenziteta zrcalnega odboja svetlobe. Parameter *endCaps* ima vrednost 0, če želimo geometrijo spremeniti iz kapsule v valj, sicer ima vrednost 1. Masko zastiranja svetlobe predstavlja vektor:

$$V_1 = (X_1, X_2, X_3, X_4), \quad (4.7)$$

kjer X_1 predstavlja razdaljo, na kateri se maska aktivira, X_2 je število zastirk enakomerno razporejenih okoli osi daljice, X_3 je prepustnost svetlobe, definirana kot vrednost z intervala $[0, 1]$, X_4 pa predstavlja mehčanje senčnega dela. Senčilniku slikovnih pik posredujemo še parameter *hasSpecular*.

Senčenje oglišč

Postopek je nekoliko bolj kompleksen kot pri točkasti ali reflektorski luči, saj je potrebno upoštevati več parametrov. Tukaj bomo omenili le bistvene korake za transformacijo geometrije:

1. Skaliranje XZ komponent oglišča s polmerom, ki ga pridobimo iz prve vrstice matrike M .
2. Razteg po Y -osi z upoštevanjem parametra *endCaps*, ki poskrbi, da se geometrija pretvori v valj, če je to potrebno.
3. Poskrbimo za ustrezno rotacijo in premik glede na dano daljico.

```
float3 position = input.Position.xyz;
float3 x = input.DataRow0.xyz;
float radius = length(x);
position.xz *= radius;
```

Izsek kode 4.8: Skaliranje XZ komponent oglišča.

```
float3 halfSegmentVector = input.DataRow1.xyz;
float halfHeight = length(halfSegmentVector);
float positionSign = sign(position.y);
float endCaps = input.DataRow3.w;
float scale = (position.y - positionSign * 0.5) * radius;
float flatten *= endCaps;
position.y = flatten + positionSign * halfHeight;
```

Izsek kode 4.9: Razteg oglišča po Y -osi.

```
float3 r1 = normalize(x);
float3 r2 = normalize(halfSegmentVector);
float3 r3 = normalize(input.DataRow2.xyz);
float3x3 rotation = {r1, r2, r3};
position = mul(position, rotation);
float3 translation = input.DataRow3.xyz;
position = float4(position + translation, 1);
```

Izsek kode 4.10: Rotacija in premik oglišča.

Senčenje slikovnih pik

Neonska luč je predstavljala nekakšen eksperimentalni del diplomskega dela, saj ni običajen tip luči, ki bi ga pogosto našli v realno-časovnih aplikacijah. Zato bomo nekoliko bolj podrobno opisali tudi postopek senčenja slikovnih pik. Preden opišemo postopek, naj omenimo, da se takšen tip osvetlitve pogosto simulira z uporabo množice točkastih luči, ki pa povzročijo neenakomerno ali premočno osvetlitev. V kolikor nam uspe zadovoljiva enakomerna

osvetlitev z razpršenimi odboji, pa nam zrcalni odboji svetlobe običajno razkrijejo točkasto naravo implementacije. Rezultat naše rešitve je enakomerna osvetlitev z razpršenimi odboji, hkrati pa ponuja zvezen izgled zrcalnega odboja.

Najprej je bilo potrebno zagotoviti ustrezno osvetlitev z razpršenimi odboji svetlobe. Sprva smo se tega problema lotili naivno, z iskanjem najbližje točke na daljici ter uporabo *Lambert*-ovega modela osvetljevanja [29]. Rezultat ni bil posebno slab, vsekakor pa se postopek ni najbolje izkazal na površinah, katerih vektor normale je vzporeden premici, na kateri leži daljica luči. Nato smo poskusili z nekoliko bolj naprednim pristopom, kjer smo izračunali delež daljice, ki vpliva na osvetlitev točke. Vendar pa se je kmalu izkazalo, da bi takšen postopek računsko preveč obremenil senčilnik, zato smo ga opustili in se vrnili nazaj k enostavnejšemu pristopu, kjer na točko vpliva le najbližja točka na daljici. Ker se je naprednejši pristop izkazal za neuporabnega, pristop z *Lambert*-ovim modelom osvetljevanja pa nam ni ustrezal, smo vedeli, da se bomo morali zadovoljiti le s približkom osvetlitve ter uporabiti nekaj trikov, da bi naredili osvetlitev uporabno v večini primerov. Tako smo prišli do rešitve, ki uporablja ukrivljeni vektor normale. Ukrivljeni vektor normale dobimo tako, da vektorju normale prištejemo vektor, ki kaže proti najbližji točki na daljici. Velikost prištetega vektorja je odvisna od oddaljenosti daljice. Tako simuliramo delež daljice, ki prispeva k osvetlitvi. Seveda smo pobrskali tudi po spletu ter ugotovili, da nekateri uporabljajo podoben pristop za površinske luči [28].

Drugi del osvetlitve je bil usmerjen v implementacijo zrcalnih odbojev svetlobe, kar se je izkazalo za težjo nalogo kot razpršeni odboji, saj le uporaba postopka *Blinn-Phong*-ovega modela osvetljevanja za izračun zrcalnih odbojev ni ustrezala izgledu, za katerega smo si prizadevali. Preko poskusov smo prišli do rešitve, kjer vrednost zrcalnega odboja svetlobe izračunamo tako, da poiščemo najbližji točki na daljici in poltraku zrcalnega odboja. Pri iskanju

```
float3 lightSegment = input.LightEnd2.xyz - input.LightEnd1.xyz;
float segmentLength = length(lightSegment);
float3 segmentDirection = normalize(lightSegment);

//closest point on line segment
float3 localSegmentStart = input.LightEnd1.xyz - worldPosition;
float t = dot(segmentDirection, -localSegmentStart);
t *= t > 0;
t = min(t, segmentLength);
float3 clp = input.LightEnd1.xyz + t * segmentDirection;
float3 localLightPosition = clp - worldPosition;
float3 cp = normalize(localLightPosition);

float cpLength = length(localLightPosition);
float radius = input.LightColorRadius.w;
float h = saturate(cpLength / radius);
float3 bentNormal = normalize(normal + cp * (1 - h * h) * 0.5));
float angleAttenuation = saturate(dot(cp, bentNormal));
float distanceAttenuation = 1 - h;
float attenuation = angleAttenuation * distanceAttenuation;
attenuation *= attenuation;

float3 color = input.LightColorRadius.rgb;
float4 lightColor = float4(color * attenuation, 1);
```

Izsek kode 4.11: Razpršeni odboji svetlobe neonske luči.

najbližjih točk smo izhajali iz postopka iskanja razdalje med dvema premicama [30] ter nekoliko optimizirali postopek z upoštevanjem specifičnosti našega primera. Končna rešitev sicer ni popolna, saj pod določenimi pogoji, kot je na primer skorajšnja vzporednost poltraku zrcalnega odboja s premico, na kateri leži daljica, razkrije nepravilnosti, ki jih nismo uspeli odpraviti. Končna vrednost zrcalnega odboja svetlobe upošteva še materialne lastnosti iz geometričnega medpomnilnika, vendar to v izseku kode 4.12 ni prikazano.

```

float3 V = normalize(POV - worldPosition);
float3 R = normalize(reflect(-V, normal));

//closest points
float a = dot(segmentDirection, R);
float b = dot(segmentDirection, localSegmentStart);
float c = dot(R, localSegmentStart);
float aa = 1 - a * a;
bool parallel = aa == 0;
aa += parallel; //to avoid division by zero
float sc = !parallel * (a * c - b) / aa;
float tc = !parallel * (c - a * b) / aa;

bool b1 = (sc > 0 && sc <= segmentLength);
bool b2 = (sc > segmentLength);
sc = b1 * sc + b2 * segmentLength;

//closest light sample
float3 cls = input.LightEnd1.xyz + sc * segmentDirection;
float3 L = cls - worldPosition;
tc = dot(R, L);

//closest point on reflection ray
float3 cpr = worldPosition + tc * R;

float dotLR = saturate(dot(normalize(L), R));
float dist = length(cls - cpr) / radius;
float spread = (1 + tc / radius);
float specularAmount = dotLR * saturate(1 - pow(dist, spread));

```

Izsek kode 4.12: Zrcalni odboji svetlobe neonske luči.

Tretji korak naše implementacije doda masko zastiranja svetlobe. Le-ta sicer ni potrebna in predstavlja precejšen dodatek k časovni zahtevnosti zaradi uporabe kotnih funkcij, vendar pa se izkaže kot koristna lastnost v nekaterih primerih osvetljevanja scene.

```

float4 bdp = input.BarnDoorsParam;
float array = cos(acos(dot(input.LightDirection, cp)) * bdp.y);
float bds = saturate(2 * bdp.z - array); //barn doors shadow
bool ds = cpLength > bdp.x; //distance shadow
spread = bdp.w * radius / cpLength;
float barnDoorsFactor = !ds + ds * pow(bds, spread);

output.Diffuse *= barnDoorsFactor;
output.Specular *= barnDoorsFactor;

```

Izsek kode 4.13: Maska zastiranja svetlobe neonske luči.

4.6 Končna osvetlitev

Po končanem osvetljevanju scene z vsemi lučmi je potreben le še en korak. To je združitev informacije geometričnega in osvetlitvenega medpomnilnika v končno sliko. Ta korak bi bil sicer nepotreben, če naša rešitev ne bi vsebovala moči izžarevanja, saj bi lahko informacijo združili že v postopku osvetljevanja. Končno sliko smo določili s formulo:

$$F = G_d * (L_d + G_e) + L_s, \quad (4.8)$$

kjer F predstavlja končno vrednost slikovne pike. G_d je barva objekta iz geometričnega medpomnilnika, L_d in L_s sta vrednosti razpršenih in zrcalnih odbojev svetlobe osvetlitvenega medpomnilnika, G_e pa je moč izžarevanja. Omeniti moramo, da bi bilo pravilneje upoštevati G_d že v postopku osvetljevanja. V primeru, ko določeno točko osvetli veliko število luči, se lahko zgodi, da vrednost osvetljenosti preseže definicijsko območje teksture, v katero shranjujemo rezultat. Temu se sicer lahko izognemo z uporabo prej omenjenega formata teksture *HdrBlendable*, ki shranjuje v načinu plavajoče vejice, vendar porabi več prostora. V kolikor pa uporabljamo teksture z definicijskim območjem $[0, 1]$, to posledično pripelje do zmanjšanih vrednosti, ko enkrat dosežemo vrednost 1. Kljub temu smo se odločili za format *R8G8B8A8*,

saj smo ocenili, da za potrebe diplomskega dela do takšnih primerov ne bo prihajalo. Dodatno pa to pomeni, da se množenje z vrednostjo G_d izvede le v izračunu končne osvetlitve ter tako za vsako luč prihranimo eno operacijo množenja v postopku osvetljevanja.

Poglavje 5

Sklepne ugotovitve

Odloženo upodabljanje je v zadnjih nekaj letih postalo zelo razširjeno predvsem med razvijalci iger. Kljub pomankljivostim namreč omogoča intuitiven način osvetljevanja, ne da bi bili pri tem strogo omejeni s številom luči. Primeren pa je tudi za veliko število naprednih postopkov za končno obdelavo, ki dodatno obogatijo izgled slike.

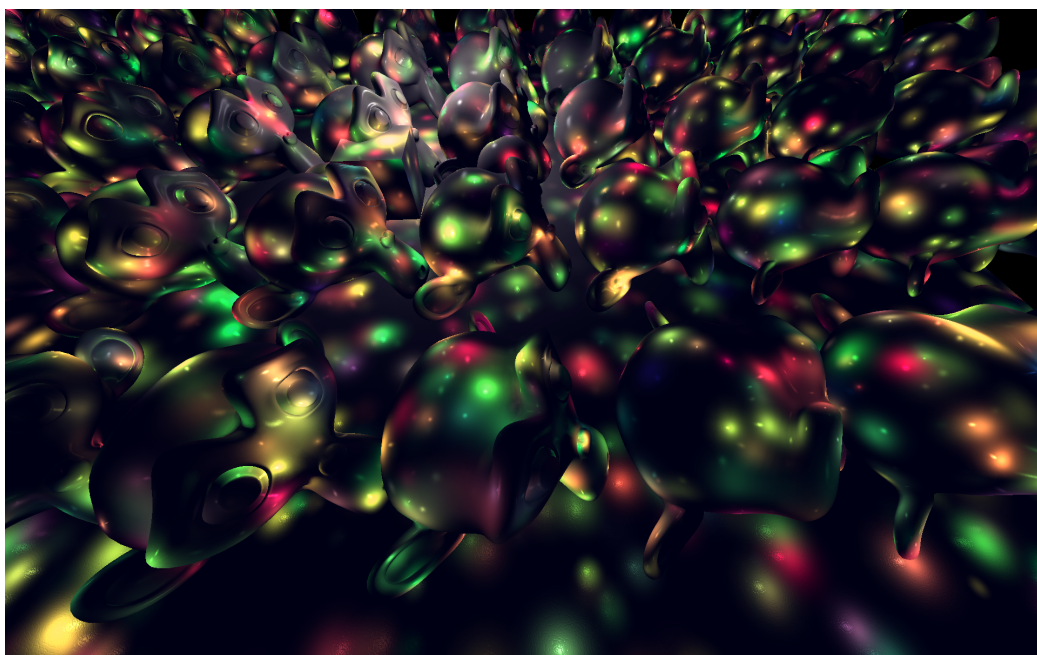
V diplomskem delu smo spoznali bistvene prednosti in slabosti odloženega upodabljanja v primerjavi s takojšnjim upodabljanjem. Predstavili smo dva najbolj tipična pristopa: odloženo senčenje in odloženo osvetljevanje. Pri implementaciji naše rešitve smo izhajali predvsem iz odloženega senčenja. Postopek smo prilagodili tako, da smo dodali osvetlitveni medpomnilnik v katerem smo razpršene in zrcalne odboje svetlobe ločili v dve teksturi. V fazi osvetljevanja nismo upoštevali barve objekta, saj smo s tem želeli nekoliko pohitriti izračun. Le-to smo upoštevali naknadno v izračunu končne slike, kjer smo dodali tudi moč izžarevanja objekta. Eksperimentalno smo se lotili tudi osvetljevanja z neonsko lučjo. Pri tem smo razvili svoj postopek osvetljevanja, predvsem kar se tiče zrcalnih odbojev svetlobe. Med samim razvojem smo poskusno implementirali tudi nekatere zanimive postopke. Kot je na primer postopek lepljenja senc, ali pa postopka za končno obdelavo *SSAO* (angl. *Screenspace Ambient Occlusion*) [31] in *RLR* (angl. *Real-Time Local*

Reflections) [32]. Tega v diplomskem delu nismo opisali, saj smo le opraskali površje teh postopkov, ki so vsak zase precej obsežno področje vredno raziskovanja.

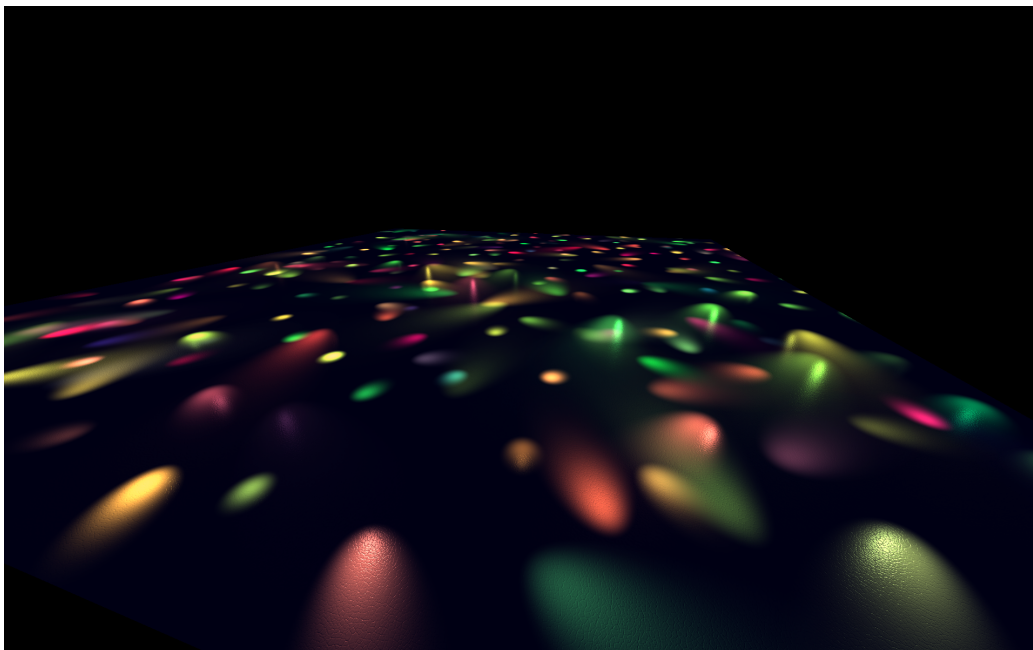
S pridobljenim znanjem in izkušnjami bi se implementacije določenih delov lotili drugače in z več razmisleka. Tako bi vektor normale v geometričnem medpomnilniku zakodirali z enim od naprednejših načinov [33]. S tem bi lahko sprostili eno od komponent teksture in/ali povečali natančnost rekonstrukcije vektorja normale. Dodatno bi lahko intenzivnost zrcalnega odboja shranili kot skalar namesto barve. Obarvanost zrcalnega odboja zaradi lastnosti materiala se pojavi le v primeru kovinskih materialov. Tako bi z dodatnim kontrolnim parametrom lahko določili ali je material kovina ali ne. V primeru kovine bi obarvanost rekonstruirali iz barve objekta. Vredno je tudi razmisliti o uporabi modela osvetljevanja *Oren-Nayar* [34] za razpršene odboje svetlobe namesto *Lambert*-ovega modela. Model *Oren-Nayar* je nekoliko bolj prilagodljiv za prikaz materialov z grobo površino, obenem pa se ga da enostavno pretvoriti v *Lambert*-ov model. Vendar pa uvedba modela *Oren-Nayar* ne sme biti povsem brez razmisleka, saj je računsko intenzivnejši od *Lambert*-ovega modela, hkrati pa je potrebno v geometrični medpomnilnik shraniti dodaten parameter. To je le nekaj možnih sprememb. Seveda pa je odločitev, katere lastnosti materiala bo vseboval geometrični medpomnilnik, predvsem odvisna od potreb končne aplikacije.

Dodatek A

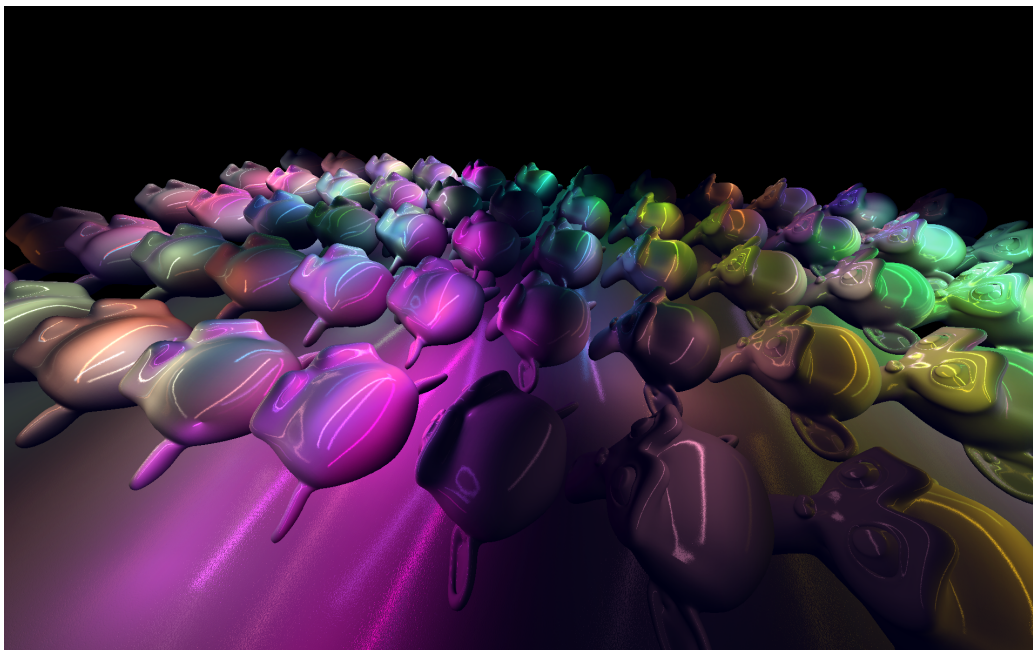
Slike nastale med razvojem



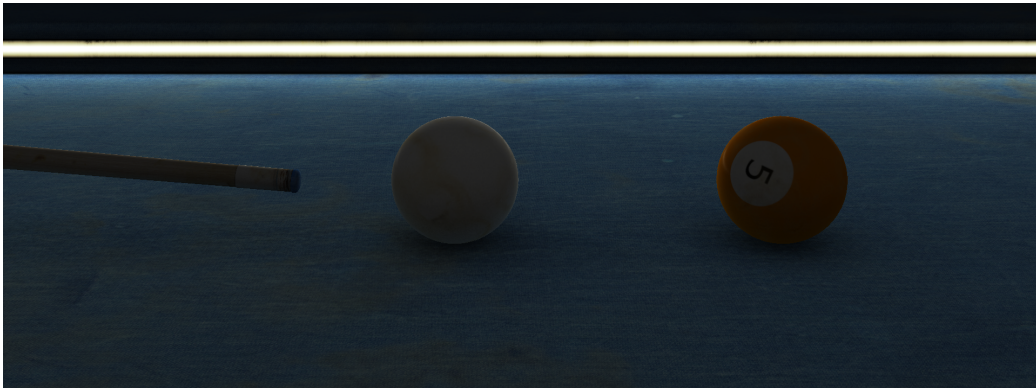
Slika A.1: Primer scene osvetljene s točkastimi lučmi.



Slika A.2: Primer scene osvetljene z reflektorskimi lučmi.



Slika A.3: Primer scene osvetljene z neonskimi lučmi.



Slika A.4: Primer objekta z definiranim izžarevanjem.

Slike

1.1	Rezultat upodabljanja	1
2.1	Geometrični medpomnilnik - oddaljenost	5
2.2	Geometrični medpomnilnik - vektor normale	6
2.3	Geometrični medpomnilnik - barva objekta	6
2.4	Geometrični medpomnilnik - intenziteta zrcalnega odboja	6
2.5	Geometrični medpomnilnik - eksponent zrcalnega odboja	6
2.6	Osvetlitveni medpomnilnik - razpršeni odboji svetlobe	7
2.7	Osvetlitveni medpomnilnik - zrcalni odboji svetlobe	7
2.8	Končna slika	7
2.9	Osvetljevanje s prednjo oziroma zadnjo ploskvijo volumna luči	11
2.10	Osvetljevanje s konstrukcijo maske iz volumna luči	11
4.1	Primer osvetlitve z ambientalno lučjo.	21
4.2	Primer osvetlitve z usmerjeno lučjo.	21
4.3	Primer osvetlitve in uporabe volumna točkaste luči	21
4.4	Primer osvetlitve in uporabe volumna reflektorske luči	22
4.5	Primer osvetlitve in uporabe volumna neonske luči	23

Tabele

4.1	Shematski prikaz geometričnega medpomnilnika.	16
4.2	Shematski prikaz osvetlitvenega medpomnilnika.	19

Izseki kode

4.1	Primer senčilnika oglišč.	18
4.2	Preslikava vektorja normale in izračun oddaljenosti.	19
4.3	Rekonstrukcija 3D lokacije.	25
4.4	Rekonstrukcija vektorja normale.	26
4.5	Skaliranje in premik oglišča.	27
4.6	Rotacija, skaliranje in premik oglišča.	29
4.7	Barva, intenziteta in kosinus največjega odklona.	29
4.8	Skaliranje XZ komponent oglišča.	31
4.9	Razteg oglišča po Y -osi.	31
4.10	Rotacija in premik oglišča.	31
4.11	Razpršeni odboji svetlobe neonske luči.	33
4.12	Zrcalni odboji svetlobe neonske luči.	34
4.13	Maska zastiranja svetlobe neonske luči.	35

Literatura

- [1] (2013) Wikipedia, "*Shadow Mapping*". Dostopno na:
http://en.wikipedia.org/wiki/Shadow_mapping
- [2] (2013) Wikipedia, "*Multisample Anti-Aliasing*". Dostopno na:
http://en.wikipedia.org/wiki/Multisample_anti-aliasing
- [3] (2013) Shawn Hargreaves, "*Deferred Shading*". Dostopno na:
<http://www.shawnhargreaves.com/DeferredShading.pdf>
- [4] (2013) Real-Time Rendering, "*Deferred Lighting Approaches*". Dostopno na:
<http://www.realtimerendering.com/blog/deferred-lighting-approaches>
- [5] (2013) Wolfgang Engel, "*Light Pre-Pass Renderer*". Dostopno na:
<http://diaryofagraphicsprogrammer.blogspot.com/2008/03/light-pre-pass-renderer.html>
- [6] (2013) Microsoft, "*Visual C# 2010 Express*". Dostopno na:
<http://www.microsoft.com/visualstudio/eng/downloads#d-2010-express>
- [7] (2013) Microsoft, "*Visual Studio*". Dostopno na:
<http://www.microsoft.com/visualstudio>
- [8] (2013) Microsoft, "*.NET*". Dostopno na:
<http://www.microsoft.com/net>

- [9] (2013) Microsoft, "*XNA 4.0*". Dostopno na:
<http://www.microsoft.com/en-us/download/details.aspx?id=23714>
- [10] (2013) Wikipedia, "*DirectX*". Dostopno na:
<http://en.wikipedia.org/wiki/DirectX>
- [11] (2013) Wikipedia, "*C#*". Dostopno na:
[http://en.wikipedia.org/wiki/C_Sharp_\(programming_language\)](http://en.wikipedia.org/wiki/C_Sharp_(programming_language))
- [12] (2013) Wikipedia, "*HLSL*". Dostopno na:
http://en.wikipedia.org/wiki/High-level_shader_language
- [13] (2013) Microsoft, "*Shader Model 3*". Dostopno na:
[http://msdn.microsoft.com/en-us/library/windows/desktop/bb509656\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/bb509656(v=vs.85).aspx)
- [14] (2013) Blender. Dostopno na:
<http://www.blender.org>
- [15] (2013) GIMP. Dostopno na:
<http://www.gimp.org>
- [16] (2013) CGTextures. Dostopno na:
<http://www.cgtextures.com>
- [17] (2013) Microsoft, "*RenderTarget Class*". Dostopno na:
[http://msdn.microsoft.com/en-us/library/microsoft.xna.framework.graphics.rendertarget\(v=xnagamestudio.31\).aspx](http://msdn.microsoft.com/en-us/library/microsoft.xna.framework.graphics.rendertarget(v=xnagamestudio.31).aspx)
- [18] (2013) Microsoft, "*RenderTarget2D Class*". Dostopno na:
[http://msdn.microsoft.com/en-us/library/microsoft.xna.framework.graphics.rendertarget2d\(v=xnagamestudio.40\).aspx](http://msdn.microsoft.com/en-us/library/microsoft.xna.framework.graphics.rendertarget2d(v=xnagamestudio.40).aspx)
- [19] (2013) Wikipedia, "*Multiple Render Targets*". Dostopno na:
http://en.wikipedia.org/wiki/Multiple_Render_Targets

- [20] (2013) Wikipedia, "*Blinn-Phong shading model*". Dostopno na:
http://en.wikipedia.org/wiki/Blinn-Phong_shading_model
- [21] (2013) Microsoft, "*SurfaceFormat Enumeration*". Dostopno na:
[http://msdn.microsoft.com/en-us/library/microsoft.xna.framework.graphics.surfaceformat\(v=xnagamestudio.40\).aspx](http://msdn.microsoft.com/en-us/library/microsoft.xna.framework.graphics.surfaceformat(v=xnagamestudio.40).aspx)
- [22] (2013) Microsoft, "*Directly Mapping Texels to Pixels*". Dostopno na:
[http://msdn.microsoft.com/en-us/library/windows/desktop/bb219690\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/bb219690(v=vs.85).aspx)
- [23] (2013) Drilian, "*Understanding Half-Pixel and Half-Texel Offsets*".
Dostopno na:
<http://drilian.com/2008/11/25/understanding-half-pixel-and-half-texel-offsets>
- [24] (2013) Wikipedia, "*Normal Mapping*". Dostopno na:
http://en.wikipedia.org/wiki/Normal_mapping
- [25] (2013) Shawn Hargreaves, "*SurfaceFormat.HdrBlendable*". Dostopno na:
<http://blogs.msdn.com/b/shawnhar/archive/2010/07/09/surfaceformat-hdrblendable.aspx>
- [26] (2013) Microsoft, "*clip HLSL intrinsic function*". Dostopno na:
[http://msdn.microsoft.com/en-us/library/windows/desktop/bb204826\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/bb204826(v=vs.85).aspx)
- [27] (2013) Wikipedia, "*Geometry Instancing*". Dostopno na:
http://en.wikipedia.org/wiki/Geometry_instancing
- [28] (2013) GameDev, "*GLSL Area Light*". Dostopno na:
<http://www.gamedev.net/topic/552315-glsl-area-light-implementation/#entry4552025>

- [29] (2013) Wikipedia, "*Lambertian Reflectance*". Dostopno na:
http://en.wikipedia.org/wiki/Lambertian_reflectance
- [30] (2013) Geomalgorithms, "*Distance Between 3D Lines & Segments*". Dostopno na:
http://geomalgorithms.com/a07-_distance.html
- [31] (2013) Wikipedia, "*Screenspace Ambient Occlusion*". Dostopno na:
http://en.wikipedia.org/wiki/Screen_space_ambient_occlusion
- [32] (2013) Gamedev, "*Real-Time Local Reflections*". Dostopno na:
<http://www.gamedev.net/blog/1323/entry-2254101-real-time-local-reflections>
- [33] (2013) Aras Pranckevičius, "*Compact Normal Storage for Small G-Buffers*". Dostopno na:
<http://aras-p.info/texts/CompactNormalStorage.html>
- [34] (2013) Wikipedia, "*Oren-Nayar Reflectance Model*". Dostopno na:
http://en.wikipedia.org/wiki/Oren-Nayar_reflectance_model