Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Luka Cindro

Simulacija in upodabljanje oblakov v realnem času

diplomsko delo
na univerzitetnem študiju

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: 01954/2013
Datum: 02.09.2013

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

Kandidat: **LUKA CINDRO**

Naslov: **SIMULACIJA IN UPODABLJANJE OBLAKOV V REALNEM ČASU**
REAL-TIME CLOUD SIMULATION AND RENDERING

Vrsta naloge: Diplomsko delo univerzitetnega študija

Tematika naloge:

V diplomski nalogi raziščite področje računalniške simulacije in upodabljanja oblakov. Podajte pregled pristopov in implementirajte rešitev, ki v realnem času simulira nastanek in gibanje oblakov, poleg tega pa te oblake dinamično upodobi z upoštevanjem smeri svetlobe.

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 Luka Cindro, z vpisno številko **63080096**, sem avtor diplomskega dela z naslovom:

Simulacija in upodabljane oblakov v realnem času

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 1. oktobra 2013 Podpis avtorja:

Zahvaljujem se mentorju doc. dr. Matiji Maroltu za nasvete pri izdelavi diplomske naloge in Janu Perčiču, ki je naredil teksture za uporabniški vmesnik.

Rad bi se zahvalil tudi staršema za podporo skozi vsa leta študija.

Kazalo

Seznam uporabljenih kratic in simbolov

CPE (*angl. Central Processing Unit*) – centralna procesna enota, osrednji del računalnika, ki obdeluje ukaze in nadzoruje ostale enote.

EBO (*angl. Element Buffer Object*) – objekt OpenGL, ki vsebuje indekse oglišč.

GLSL (*angl. OpenGL Shading Lanuage*) – programski jezik za pisanje senčilnikov.

GPU (*angl. Graphics Processing Unit*) – grafični procesor, enota namenjena upodabljanju slik na računalniku.

PCIe (*angl. Peripheral Component Interconnect Express*) – vodilo za razširitvene kartice.

VAO (*angl. Vertex Array Object*) – objekt OpenGL, ki enkapsulira stanje za upodobitev oglišč.

VBO (*angl. Vertex Buffer Object*) – objekt OpenGL, ki omogoča učinkovito shranjevanje podatkov za upodobitev oglišč.

Povzetek

Oblaki imajo pomembno vlogo v prizorih iz računalniških iger in simulatorjev, vendar sta zaradi njihovih fizičnih lastnosti tako simulacija njihovega nastajanja in premikanja kot tudi upodabljanje težavna. Problem je še toliko zahtevnejši, kadar moramo obe operaciji izvesti v realnem času z omejeno strojno opremo. V diplomskem delu predstavimo različne pristope za simulacijo in upodabljanje oblakov. V programskem jeziku C++ smo implementirali metodo za simulacijo oblakov, ki uporablja celični avtomat za njihovo hitro, a navidez turbulentno generiranje. Z uporabo programske knjižnice OpenGL smo razvili upodabljanje oblakov zasnovano na volumetričnem metanju žarkov, kjer s senčnimi žarki aproksimiramo enojno sipanje svetlobe. Z našo rešitvijo je možno v realnem času simulirati in upodobiti oblake na sodobnih računalnikih.

Ključne besede:
simulacija, upodabljanje, oblaki, celični avtomat

Abstract

Clouds play an important role in computer game and simulator scenes, but both rendering of clouds and simulation of their movement are difficult because of their physical characteristics. The problem gets even more involved when the operations have to be done under real-time constraints with limited hardware capabilities. In this thesis, we describe methods for cloud simulation and rendering. We implemented a fast method for cloud simulation based on cellular automata using the C++ programming language. We realized cloud rendering by using the OpenGL library. Our rendering method is based on volumetric ray casting and approximates single-scattering of light by traversing additional shadow rays. The proposed method can realize cloud simulation and rendering in real-time on modern computers.

Key words:
simulation, rendering, clouds, cellular automaton

Poglavje 1

Uvod

Angleški romantični slikar John Constable je znan predvsem po slikah pokrajine v okolici svojega rojstnega kraja, na večini njegovih slik pa je predvsem izrazito nebo, kateremu je zmeraj namenil veliko pozornosti. Njegova predanost slikanju oblakov je bila tako velika, da je leta 1821 in 1822 ustvaril več kot 50 slik na katerih ni bila narisana pokrajina, ampak le nebo. Oblaki so ga zanimali tudi z znanstvenega vidika – na zadnje strani slik je zapisoval stanje vremena, položaj sonca in čas slikanja.

Tudi programerji, ki se ukvarjajo z računalniško grafiko, na nek način slikajo pokrajino, sestavni del te pokrajine pa je nebo. Kljub vsem napredkom v znanosti pa oblaki še zmeraj ostajajo uganka. Njihovo gibanje je kaotično, svetlobni žarki, ki potujejo skozi njih, pa se lomijo na zanimive, a kompleksne načine. Problem postane še toliko zahtevnejši, ko moramo upoštevati omejitve sodobnih računalnikov. Za natančno simulacijo in upodabljanje oblakov bi porabili ogromno prostora in računske moči, zato moramo za praktično implementacijo poseči po poenostavitvah in optimizacijah.

Pri računalniških igrah programerji najpogosteje uporabijo najenostavnejše rešitve za izris neba. To je običajno predstavljeno s statično sliko, ki je izrisana na robu vidnega polja. Oblaki na taki sliki ne nastajajo, izginevajo, ali se premikajo, prav tako pa jih ne moremo dinamično osvetliti glede na položaj sonca. V zadnjih letih je napredek v strojni opremi omogočil uporabo

Slika 1.1: John Constable - Cloud Study (1821).

tudi drugih metod za upodabljanje oblakov, a kljub temu njihova uporaba v sodobnih računalniških igrah pa še zmeraj ni pogosta.

V diplomskem delu se seznanimo z metodami, ki se uporabljajo za simulacijo in upodabljanje oblakov. Implementirali smo program, ki v realnem času simulira nastanek ter gibanje oblakov, poleg tega pa te oblake dinamično izriše glede na smer svetlobe. Za simulacijo oblakov smo implementirali Dobashijevo metodo s celičnim avtomatom, za izris oblakov pa napisali senčilnik, ki upošteva en odboj žarkov svetlobe.

Poglavje 2

Definicija problema

2.1 Fizične lastnosti oblakov

Oblaki so viden skupek majhnih kapljic in ledenih kristalov vode ter ostalih snovi, ki se nahajajo v zraku. Nastanejo tam, kjer je zrak nasičen z vodno paro ali kot posledica ohlajanja zraka. Z višino tlak pada, tako da se zrak, ki se dviga, razširi in ohladi. Ko vodna para doseže dovolj nizko temperaturo in visoko gostoto, se kondenzira (preide iz plinastega v kapljevinasto agregatno stanje) in nastane oblak. Ko se nasičenost še bolj poveča, kaplje dobijo dovolj veliko hitrost, da zapustijo oblak, in nastanejo padavine.

Oblake delimo na več vrst. *Cumulus*, prikazani na sliki 2.1, so puhasti oblaki z jasnimi robovi, *stratus* so nizki brezoblični oblaki, *cirrus* pa visoki tanki oblaki. Poleg teh glavnih skupin se za klasifikacijo uporabljajo še predpone, kot so *nimbus* za padavinske oblake, *cumulo* za kompleksne nevihtne oblake, *alto* za oblake na nadmorski višini med 2 in 6 kilometra in *cirro* za oblake na nadmorski višini 6 kilometrov in več.

2.2 Sipanje svetlobe v oblakih

Za realistično upodobitev oblakov moramo razumeti interakcijo svetlobe s snovjo, ki jih sestavlja. Oblaki so večinoma sestavljeni iz mnogo majhnih

Slika 2.1: Oblaki tipa cumulus.

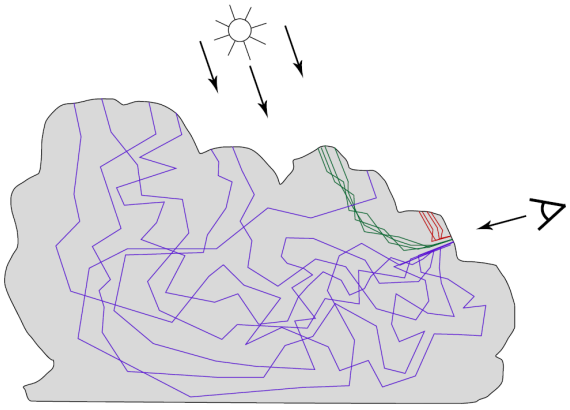
delcev strnjene vode, žarek svetlobe pa lahko obravnavamo kot tok fotonov. Ko foton zadene drug delec, se ali absorbira, ali odbije. Pri oblakih prevladujejo odboji fotonov. Ker se fotoni lahko odbijejo v različne smeri, se žarek, ki ga usmerimo v nek delec, odbije v več smeri. Ta pojav imenujemo *sipanje svetlobe*.

Verjetnost, da se foton odbije v neko smer, je odvisna od lastnosti delca, v katerega se je zaletel. Te verjetnosti lahko za neko snov opišemo z *fazno funkcijo*, ki za podan vpadni kot poda verjetnostno porazdelitev odbojnega kota fotona.

Kadar so delci veliko manjši od valovne dolžine svetlobe, lahko odboje fotonov na njih približno izračunamo z Rayleighovim sipanjem [8]. Delci plinov v atmosferi so taki, tako da lahko z Rayleighovim sipanjem pojasnimo pojave, kot sta modra barva neba podnevi in rdeča barva neba ob sončnem vzhodu in zahodu. Sipanje svetlobe v snoveh z večjimi delci bolje opisuje Mievo sipanje [8]. Ker je fazno funkcijo Mievoga sipanja težko izračunati, se namesto nje pogosto uporablja njene približke, kot je Henyey-Greensteinova fazna funkcija:

$$P_{HG}(\phi) = \frac{1}{4\pi} \frac{1 - g^2}{(1 + g^2 - 2g\cos\phi)^{\frac{2}{3}}}, \tag{2.1}$$

kjer je ϕ vpadni kot svetlobe, g pa paremeter, ki določa, koliko svetlobe se



Slika 2.2: Različne poti fotonov v oblaku [3].

odbije naprej in nazaj. Henyey-Greensteinova fazna funkcija dobro opisuje sipanje svetlobe v oblakih, kjer je velikost delcev približno tako velika, kot valovna dolžina svetlobe. Za razliko od Rayleigove in Mieve funkcije ne upošteva valovne dolžine, tako da je sipanje svetlobe enako za vse barve.

Oblaki so v primerjavi z večino drugih objektov, ki jih upodabljamo v računalniških programih, zelo redki. Foton v povprečju prepotuje okoli 20 metrov, preden zadane kakšen drug delec [2]. Zaradi tega in zaradi karakteristik fazne funkcije delcev oblaka, obstaja velika verjetnost, da se foton, ko vstopi v oblak, večkrat odbije, tako kot je prikazano na sliki 2.2. Ta pojav imenujemo *večkratno sipanje* (*angl. multiple scattering*). Kadar se foton odbije samo enkrat, ali kadar pri izračunih upoštevamo samo en odboj, to imenujemo *enkratno sipanje* (*angl. single scattering*). Zaradi večkratnega sipanja imajo oblaki belo barvo in navidez difuzno površino [8].

2.3 Oblaki v računalniških programih

V programih s področja računalniške grafike je pogosto potrebno izrisati oblake. Najpogosteje se pojavljajo v računalniških igrah in programih za

izdelavo filmskih vizualnih efektov, poleg tega jih pa najdemo tudi v vojaških simulatorjih, še posebej pomembni so pa pri simulatorjih letenja.

Pristop k izrisu oblakov se bistveno razlikuje glede na tip programa. Računalniške igre in simulatorji so *realno časovni programi*, kar pomeni, da morajo proizvesti rezultat v okviru časovnih omejitev. Da pri animaciji, v kateri gibanje ni zamegljeno, ni možno opaziti nezveznosti, mora program prikazati 30 ali celo 60 sličic na sekundo [1]. To pomeni, da moramo pri realno časovnih programih, ki prikažejo 60 sličic na sekundo, vsako sličico upodobiti v največ 17 milisekundah, medtem ko lahko orodja za izdelavo filmskih vizualnih efektov za izris vsake sličice filma porabijo več ur. Pri slednjih se izračuni delajo na več računalnikih hkrati, računalniške igre pa morajo delovati na mnogo manj zmogljivih sistemih. Realno časovni programi to ogromno razliko v računski moči premostijo s številnimi optimizacijami in poenostavitvami. Rezultat je tako manj vizualno prepričljiv, vendar je za njegov izračun potrebno veliko manj časa.

Izbira metode za simulacijo in izris oblakov je odvisna tudi od tega, koliko oddaljeni bodo od virtualne kamere. Če so oblaki izrisani v daljavi, jih lahko nadomestimo s statičnimi slikami, kadar pa se kamera oblakom lahko približa, ali celo gre skozi njih, jih pa moramo dinamično spreminjati glede na zorni kot, da proizvedemo občutek globine.

Pri večini računalniških iger hočejo ustvarjalci čim bolj vizualno prepričljivo prikazati okolico – tako, da je ta podobna okolici, ki bi jo lahko posneli s kamero. Obstajajo pa tudi take igre, kjer se ustvarjalci temu namerno izogibajo. Namesto da bi objekte upodabljali čimbolj realistično, posnemajo risbe, ilustracije, risanke ali druge umetniške stile. Tudi oblaki pri takih igrah so drugače upodobljeni, tako da bi za njihovo upodobitev izbrali drugačno metodo.

Zmogljivost grafičnih kartic še zmeraj hitro raste. Z večjo računsko močjo postajo dostopni novi postopki na področju realno časovno računalniške grafike. Med njimi je tudi volumetrično upodabljanje, ki ga lahko uporabimo za upodabljanje oblakov.

2.4 Cilj diplomskega dela

Želimo ustvariti program, ki bo prikazal oblake, ki se bodo med njegovim izvajanjem spreminjali. Njihova upodobitev bo dinamična, torej odvisna od položaja vira svetlobe in ostalih parametrov. Osredotočili se bomo na oblake tipa cumulus, ki imajo relativno definirane robove in puhast videz. Prizadevali si bomo upodobiti čim bolj realistične oblake, na pa stilizirane oblake, kot jih najdemo na primer v risankah.

Pri simulaciji oblakov se hočemo izogniti modeliranju. Uporabnik programa bo izbral parametre, simulacija pa bo ustvarila oblake, ki se bojo dinamično spreminjali. Pri tem ni potrebno, da je fizikalno točna, le oblaki morajo imeti približno tako obliko, kot jo imajo v resnici. Za simulacijo tudi ni potrebno da je hitra, saj se oblaki običajno zelo počasi premikajo.

Pri upodabljanju tudi ne bomo upoštevali vseh fizičnih lastnosti oblakov, saj bi to bilo računsko zelo zahtevno. Pri senčenju se bomo omejili na enojne odboje svetlobnih žarkov. Želimo dobiti nekatere vizualne efekte, kot so temnejša spodnja stran oblakov, polprozornost in svetla obroba oblakov, kadar gledamo proti soncu.

Pri implementaciji programa in izbiri metod je pomembna omejitev, da mora program teči v realnem času. Želimo, da izriše vsaj 30 sličic na sekundo na sodobnih računalnikih.

Poglavje 3

Sorodna dela

Čeprav ima veliko novih računalniških iger odprte prostore, v katerih je nebo pomemben del vizualne podobe, se pri večini za upodabljanje neba uporablja eno izmed izmed enostavnejših metod. Prav tako je dinamična simulacija oblakov le redko implementirana, a vseeno je simulacija tekočin zelo aktivno področje znanstvenega raziskovanja.

Na področju simulacije in upodabljanja oblakov je bilo narejenih že nekaj raziskav, poleg tega si pa lahko pomagamo z znanjem s podobnih področij. Obstaja kar nekaj rešitev, ki se med seboj razlikujejo v računski zahtevnosti in prepričljivosti. Poleg tega pa postajajo z napredkom v računski zmogljivosti in arhitekturi grafičnih kartic nove metode primerne za implementacijo v realno časovnih programih.

3.1 Simulacija

Računalniške igre pogosto uporabljajo fizikalni pogon, ki simulira fizikalno interakcijo med telesi. Pri tem je ta običajno omejen na toga telesa, saj je simulacija mehkih teles in tekočin veliko bolj težavna. Med simulacijo tekočin spada tako simulacija kapljevin kot tudi plinov, tako da nas za implementacijo našega programa zanima prav ta del fizikalnih pogonov.

Metode simulacije tekočin izvirajo iz področja imenovanega *računalni-*

ška dinamika tekočin, katero zajema računske metode, ki rešujejo probleme pretoka tekočin, uporabljajo pa se na področjih, kot so aviacija, pomorstvo, medicina in meteorologija. Metode simulacije tekočin se uporabljajo v računalniški grafiki in se razlikujejo z metodami računalniške dinamike tekočin v tem, da morajo biti rezultati le vizualno prepričljivi, ne pa tudi fizikalno točni.

3.1.1 Navier-Stokesove enačbe

Navier-Stokesove enačbe so enačbe, ki opisujejo gibanje stisljive, viskozne tekočine. Enačbe običajno opišemo s sistemom nelinearnih parcialnih diferencialnih enačb. V vektorskem zapisu je ta sistem:

$$\rho\left(\frac{\partial \mathbf{v}}{\partial t} + \mathbf{v}\cdot\nabla\mathbf{v}\right) = -\nabla p + \nabla\cdot\mathbf{T} + \mathbf{f}, \tag{3.1}$$

kjer je ρ gostota toka, $\mathbf{v}$ hitrost toka, p pritisk, T komponenta tenzorja napetosti in $\mathbf{f}$ vsota sil, ki delujejo na tekočino.

Pri tekočinah pogosto opazimo *turbulenco* - navidez kaotično obnašanje tekočine. Domneva se, ni pa dokazano, da Navier-Stokesove enačbe pravilno opisujejo gibanje tako turbulentnih tekočin, kot tudi tekočin z visoko viskoznostjo. Ker je sistem linearnih enačb nelinearen, je pogosto zelo težko najti analitično rešitev [8]. Tudi numerično reševanje Navier-Stokesovih enačb je težavno, saj je za natančne rezultate zelo računsko zahtevno. Zaradi tega se v računalniški dinamiki tekočin uporabljajo hitrejši računski modeli za reševanje Navier-Stokesovih enačb, kot sta RANS (*angl. Raynolds-Averaged Navier-Stokes equations*) in LES (*angl. Large Eddy Simulation*).

3.1.2 Lattice Boltzmannove metode

Lattice Boltzmannove metode namesto Navier-Stokesove enačbe rešujejo diskretno Boltzmannovo enačbo, pri kateri je snov razdeljena na majhne (namišljene) delce, med katerimi se zaznava trčenje. Boltzmannova enačba opisuje

statistično obnašanje termodinamičnega sistema in jo lahko zapišemo kot:

$$\partial_t f + \xi \cdot \nabla_x = Q(f), \tag{3.2}$$

kjer je $f(t, x, \xi)$ funkcija, ki opisuje gostoto delcev ob času t, na poziciji x, s hitrostjo ξ, Q pa omejen podprostor, v katerem preverjamo trčenja [4].

Ker Lattice Boltzmannove metode uporabljajo delce, ki imajo lokalen vpliv, jih je možno učinkovito paralerizirati. Poleg tega je z njimi možno modelirati interakcijo tekočine in kompleksnih objektov. Tako kot metode, ki rešujejo Navier-Stokesove enačbe, so tudi Lattice Boltzmannove metode računsko zahtevne.

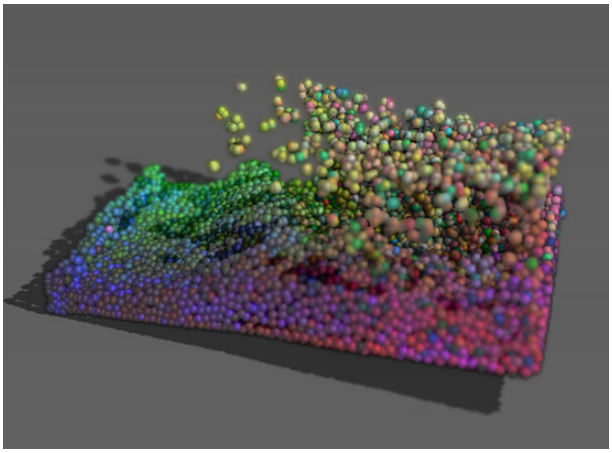
3.1.3 Hidrodinamika zglajenih delcev

Tako kot pri Lattice Boltzmannovi metodi, opisani v razdelku 3.1.2, je pri hidrodinamiki zglajenih delcev (*angl. smoothed particle hydrodynamics*) snov razdeljena na delce, kot je prikazano na sliki 3.1. Med temi delci poteka simulacija skladno z Newtonovomi zakoni gibanja. Da dobimo fizično lastnost delca na poljubnem položaju, z jedrno funkcijo zgladimo to fizično lastnost bližnjih delcev. Najpogosteje se za jedrno funkcijo uporabi kubični zlepek ali Gaussovo krivuljo.

Hidrodinamiko zglajenih delcev se uporablja na različnih področjih, kot sta astrofizika in balistika, vse pogosteje se jo pa uporablja tudi za simulacijo tekočin. Kadar delcev ni veliko, je možno simulirati premike delcev v realnem času, a z manjšim številom delcev pada ločljivost in natančnost simulacije [10]. Kljub temu je hidrodinamika zglajenih delcev primerna za računalniške igre, kjer natančnost simulacije ni zelo pomembna.

3.1.4 Celični avtomati

Celični avtomat je diskreten model, ki ga sestavlja mreža *polj* (*celic*). Vsako polje je pri določenem času v nekem stanju iz končne *množice stanj* (pogosto sta stanji samo dve, npr. 0 in 1). Čas je diskreten, prehode polj med stanji



Slika 3.1: Simulacija hidrodinamike zglajenih delcev s programom FLUIDS.

pa določa *funkcija prehodov stanj*. Ta funkcija je najpogosteje neodvisna od časa in enaka za vsa polja. Poleg tega je podano še začetno stanje polj.

Celični avtomati so zanimivi, saj je tudi s preprostimi pravili možno dobiti navidez kaotično oziroma psevdonaključno delovanje. Med najbolj znanimi celičnimi avtomati je *igra življenja*, ki jo je leta 1970 objavil britanski matematik John Horton Conway [11]. S primernim začetnim stanjem lahko pri igri življenja dobimo zanimive vzorce, ki ostanejo aktivni dolgo časa.

S celičnimi avtomati lahko modeliramo številne naravne pojave, kot so dinamika plinov in tekočin, širjenje požarov in kristalizacija. Uporabljajo se tudi na področju kriptografije in umetne inteligence.

3.2 Upodabljanje

Tako kot pri simulaciji tekočin imamo tudi pri upodabljanju polprozornih volumetričnih oblik več metod, ki se med seboj razlikujejo v računski zahtevnosti in prepričljivosti. Enostavnejše metode uporabljajo prej izračunane slike, da se izognejo veliki porabi pomnilnika in zahtevnemu računanju. Pri ostalih metodah pa oblake modeliramo ali proceduralno generiramo in nato izrišemo z bolj zahtevnim senčenjem.

Računalniška grafika se običajno ukvarja z upodabljanjem mrež poligonov. Oblake težko modeliramo z mrežami poligonov, saj so polprozorni, poleg tega je pa njihova gostota nehomogena. Najpogosteje take objekte upodabljamo z uporabo slik (*angl. image-based rendering*). Te slike so največkrat dvodimenzionalne, različne pristope k upodabljanju oblakov z uporabo takih slik pa opisujemo v razdelku 3.2.1. Namesto njih lahko uporabimo tudi trodimenzionalne slike, kjer oblak opišemo z diskretnim 3D poljem, kjer vsaka celica predstavlja gostoto oblaka za ta položaj. Z upodabljanjem takih polj se ukvarja področje imenovano *upodabljanje volumnov*, ki je najpogosteje uporabljeno v računalniški tomografiji in filmski industriji, prodira pa tudi na področje računalniških iger. Metodi s tega področja, ki se najpogosteje uporabljata za izrisovanje oblakov in ostalih polprozornih objektov, sta opisani v razdelkih 3.2.2 in 3.2.3. Ostale volumetrične metode so podrobno opisane v [7].

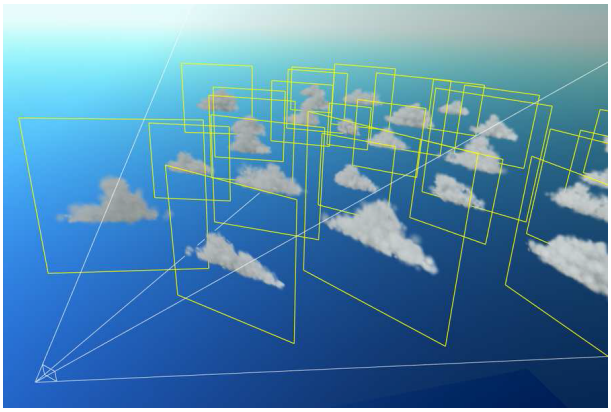
3.2.1 Upodabljanje z dvodimenzionalnimi slikami

Poglavitna prednost uporabe slik v računalniški grafiki je, da ni potrebno uporabiti veliko število poligonov za upodobitev kompleksnega predmeta. To se še posebej pozna za oddaljene predmete, kjer bi moralo biti število oglišč zelo veliko [1]. Zaradi tega se v 3D računalniških igrah zelo pogosto okoli igralca izriše kocko, na vsaki stranici te kocke pa slika (*angl. skybox*). Na teh slikah so narisani predmeti, do katerih igralec ne more priti, kot so oddaljene gore, sonce in oblaki. Tekstura za tako kocko je ponavadi sestavljena iz šestih kvadratov, kot je prikazano na sliki 3.2. Ko se kamera premika, se (tipično) kocka premika skupaj s kamero, tako da ima igralec občutek, da so predmeti izrisani na njej zelo oddaljeni. Slabost takega pristopa je, da so te predmeti povsem statični. Oblaki niso animirani, prav tako pa osvetljenost predmetov ni odvisna od položaja luči v sceni.

S slikami lahko upodobimo tudi predmete, ki se nahajajo bližje igralcu. V tem primeru moramo paziti, da delujejo prepričljivo, tudi kadar se jim igralec približa, jih obkroži, oziroma celo gre skozi njih. Kot je prikazano na

Slika 3.2: Slika okolice generirana s programom Terragen.

sliki 3.3, so te slike pogosto obrnjene proti kameri (*angl. billboarding*). Tak pristop k izrisovanju oblakov je uporabila Wang [16]. Prozornost oblakov spreminja glede na oddaljenost igralca do njih, zato da deluje, kot da oblaki nastajajo in izginevajo. Za optimizacijo več oddaljenih oblakov shrani v eno teksturo, ki jo posodobi na vsake par sličic. Ellinas in Stürzlinger uporabita več gnezdenih elips, ki so bolj prozorne na robovih, in Perlinov šum za generiranje in upodobitev oblakov [6].

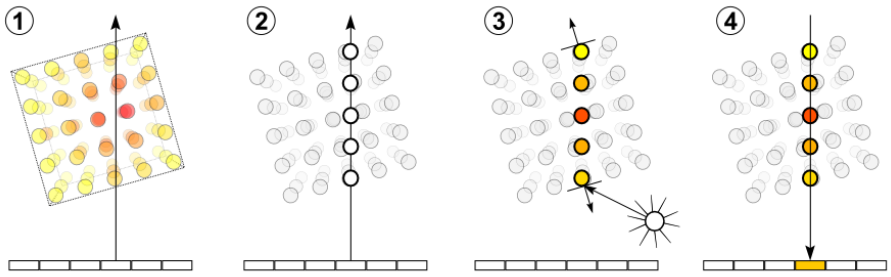


Slika 3.3: Upodabljanje oblakov s slikami, obrnjenimi proti kameri [8].

2D teksturo lahko dinamično generiramo med izvajanjem programa, tako da v njo shranimo upodobitve oddaljenih 3D objektov (*angl. impostor*). Tako teksturo nato uporabimo za več enakih objektov ali za par zaporednih sličic, s čimer optimiziramo upodabljanje [1]. Tak pristop je posebej uporaben za upodabljanje kompleksnih oddaljenih predmetov. Ko kot med kamero in položajem, za katerega je bila tekstura narejena, postane prevelik, se jo nadomesti z drugo. Med prehodom se lahko sliki interpolira, da je ta manj opazen. Tak pristop ni uporaben za predmete, ki hitro spreminjajo svojo obliko ali usmeritev. Harris je to metodo uporabil za učinkovit realno-časoven izris oblakov [8].

3.2.2 Volumetrično metanje žarkov

Volumetrično metanje žarkov je metoda, ki na podlagi 3D polja podatkov izriše 2D sliko. Na sliki 3.4 so prikazani njeni osnovni koraki:



Slika 3.4: Volumetrično metanje žarkov.

1. **Metanje žarka:** Za vsak piksel končne slike je poslan žarek (premica) skozi polje podatkov. Tipično izračunamo točki, kjer žarek seka ovojnico tega polja.

2. **Vzorčenje:** Točke, ki ležijo na žarku, so vzorčene. Te točke so tipično ekvidistančne in se nahajajo znotraj ovojnice polja podatkov. Ker se položaji točk na žarku ne ujemajo povsem z položaji točk v polju, se bodisi izbere najbližjo točko ali pa interpolira bližnje točke.

3. **Senčenje:** Za vsako vzorčeno točko je izračunana neka vrednost. Na izračunano vrednost lahko vplivajo različni podatki, kot so vzorčena vrednost, položaji virov svetlobe, lega na žarku itd.

4. **Kompozicija:** Na podlagi senčenih vrednosti je izračunana končna vrednost piksla. Kompozicija tipično poteka od najbolj oddaljene točke k najbližji, tako da točke, ki so bližje, delno prekrijejo točke, ki so bolj oddaljene.

3.2.3 Projiciranje vokslov

Projiciranje vokslov (*angl. splatting*) je metoda upodabljanja volumetričnih podatkov, kjer se na mestu vsakega voksla izriše majhno polprozorno sliko (*angl. splat*). Ta slika je tipično krog, ki je bolj prozoren na robovih, namesto kroga pa se lahko uporabi tudi druge oblike [1]. Voksle se za izris razvrsti po oddaljenosti, tako da se najbolj oddaljenega izriše najprej. Ko so vsi voksli izrisani, se običajno uporabi Gaussov ali podoben filter, da zakrije presledke med slikami. Z metodo lahko upodabljamo tudi za neprozorne predmete. Rusinkiewicz in Levoy sta metodo uporabila za upodabljanje zelo podrobnih 3D predmetov [14]. Dobashi je uporabil tak pristop za upodabljanje oblakov [5]. V času njegovega dela je upodobitev vsake sličice trajala par sekund.

Poglavje 4

Implementacija simulacije

Izmed metod, opisanih v razdelku 3.1, smo izbrali celični avtomat, saj je računsko nezahteven in tudi s preprostimi pravili lahko dobimo navidezno kompleksne rezultate. Pravila prehodov med stanji celic avtomata za simuliranje oblakov, na katerih sloni naša implementacija, je najprej opisal Nagel in nato razširil Dobashi [12, 5].

4.1 Pravila simulacije

Stanje celičnega avtomata opišemo s tridimenzionalno mrežo polj. Vsako polje ima tri logične spremenljivke:

Vlaga (*hum*) je v stanju 1, kadar je v polju dosti vlage, da nastane oblak.

Funkcija faznega prehoda (*act*) je v stanju 1, kadar je možno, da polje preide iz stanja $hum = 1$ v stanje $cld = 1$.

Oblak (*cld*) je v stanju 1, kadar je v polju prisoten oblak.

Pravila prehodov med stanji celičnega avtomata, kot jih je opisal Nagel, so opisana s sledečimi enačbami:

$$\begin{aligned} hum(i,j,k,t_{i+1}) &= hum(i,j,k,t_i) \wedge \neg act(i,j,k,t_i) \\ act(i,j,k,t_{i+1}) &= \neg act(i,j,k,t_i) \wedge f_{act}(i,j,k) \\ cld(i,j,k,t_{i+1}) &= cld(i,j,k,t_i) \vee act(i,j,k,t_i), \end{aligned} \tag{4.1}$$

kjer je $f_{act}(i,j,k)$ logična spremenljivka, ki jo izračunamo na podlagi sosednjih polj celičnega avtomata:

$$\begin{aligned} f_{act}(i,j,k) = & act(i-2,j,k,t_i) \vee act(i-1,j,k,t_i) \vee \\ & \vee act(i+1,j,k,t_i) \vee act(i+2,j,k,t_i) \vee \\ & \vee act(i,j-2,k,t_i) \vee act(i,j-1,k,t_i) \vee \\ & \vee act(i,j+1,k,t_i) \vee act(i,j+2,k,t_i) \vee \\ & \vee act(i,j,k-2,t_i) \vee act(i,j,k-1,t_i) \vee \\ & \vee act(i,j,k+1,t_i). \end{aligned} \tag{4.2}$$

Pri takemu celičnemu avtomatu je potek simulacije zelo odvisen od začetnega stanja. Poleg tega oblaki, simulirani s takim celičnim avtomatom, nikoli ne izginejo. Kadar logična spremenljivka *cld* nekega polja zasede vrednost 1, nikoli več ne preide v vrednost 0. Prav tako kadar *hum* zasede vrednost 0, ne more več preiti v vrednost 1.

Simulacijo je razširil Dobashi. Pravila za vrednost logičnih spremenljivk *hum*, *act* in *cld* je dopolnil tako, da so odvisna od naključno generiranih števil. V ta namen je uvedel parameter p_{ext}, ki predstavlja verjetnost izginotja oblaka. Za vsako polje, katerega logična spremenljivka *cld* je enaka 1, vsak korak generirano naključno število rnd ($0 \leq rnd \leq 1$). Kadar je za to polje $rnd < p_{ext}$, se vrednost *cld* spremeni v 0. Tudi s to izboljšavo je simulacija težavna, saj kadar *cld* preide iz 1 v 0, ne more preiti nazaj v 1. Zaradi te težave je dodal še parametra p_{act} in p_{hum}, ki z naključno vrednostjo rnd spreminjata vrednosti logičnih spremenljivk *act* in *hum*. Po vsakem koraku simulacije, opisane z enačbami (4.1), so vrednosti logičnih spremenljivk

popravijo s sledečimi pravili:

$$\begin{aligned} hum(i,j,k,t_{i+1}) &= hum(i,j,k,t_i) \vee (rnd < p_{hum}(i,j,k,t_i)) \\ act(i,j,k,t_{i+1}) &= act(i,j,k,t_i) \vee (rnd < p_{act}(i,j,k,t_i)) \\ cld(i,j,k,t_{i+1}) &= cld(i,j,k,t_i) \wedge (rnd < p_{ext}(i,j,k,t_i)), \end{aligned} \tag{4.3}$$

kjer je rnd naključno število z intevala $[0,1]$, generirano v vsakem koraku za vsako polje posebaj.

Da bi simuliral še premikanje oblakov zaradi vetra, je Dobashi uvedel zamikanje celic po vsakem koraku simulacije. Velikost zamika je odvisna od parametra hitrosti $v(z_k)$. Hitrost je odvisna od z koordinate, saj veter pogosto močnejše piha na višji višini. Ker je smer vetra na neki višini enaka za vsa polja, s takim modelom ne simuliramo bolj naprednih pojavov, kot je turbulenca.

Preden upodobimo polje, ki predstavlja oblake, ga moramo še zgladiti, da namesto binarnih vrednosti 0 in 1 vsebuje vrednosti z zveznega intervala $[0,1]$. Dobashi v ta namen uporabi kompleksno Wyvillovo funkcijo. V naši implementaciji namesto tega z jedrno funkcijo preidemo polje in za vsako celico izračunamo povprečno vrednost elementov pod jedrom. Tak pristop je enostavnejši in hitrejši, a še zmeraj proizvede dovolj dobre rezultate.

4.2 Upravljanje položaja oblakov

V razdelku 4.1 smo opisali parametre p_{ext}, p_{hum} in p_{act}, s katerimi modeliramo nastajanje oblakov. Dobashi uporablja elipsoide, s katerimi upravlja položaj, velikost in premikanje oblakov. Te parametri so izbrani za središča elipsoidov. Z večanjem razdalje od središča elipsoida p_{hum} in p_{act} padata in p_{ext} raste.

Pri naši implementaciji simulacije imamo oblake (elipsoide) različnih velikosti. Da se izognemo anomalijam, ki nastanejo, kadar upoštevamo samo najbližji oblak, za vsako polje poiščemo vrednost mds, ki predstavlja najmanjše razmerje med oddaljenostjo tega polja in središčem oblaka ulomljeno z velikostjo oblaka. Ker hočemo dobiti relativno ostre robove oblakov, izračunamo

vrednost d z eksponentno funkcijo:

$$d = e^{-mds}, \tag{4.4}$$

kjer je e parameter, ki določa, koliko razpršeni so oblaki.

Ko izračunamo vrednost d za neko polje, lahko parametre p_{ext}, p_{hum} in p_{act} skaliramo z naslednimi enačbami:

$$\begin{aligned} sp_{hum} &= p_{hum} * (1 - d) \\ sp_{act} &= p_{act} * d \\ sp_{ext} &= p_{ext} * d. \end{aligned} \tag{4.5}$$

Po izračunu še poskrbimo, da skalirane vrednosti sp_{ext}, sp_{hum} in sp_{act} zasegajo vrednosti iz intevala $[0, 1]$.

Elipsoide lahko določi grafični oblikovalec ali pa jih ustvarimo programsko. V našem programu jih generiramo tako, da za središče oblaka izberemo naključni položaj v polju celičnega avtomata (oddaljena od roba za neko minimalno razdaljo), za velikost pa naključno vrednost s prej izbranega intervala. Tudi elipsoide premikamo skupaj z vetrom.

4.3 Večnitno izvajanje

Korak simulacije je relativno dolga operacija, saj v vsakem koraku program preide celotno tridimenzionalno polje. Pri testiranju programa smo uporabljali mrežo celičnega avtomata velikosti $128 \times 128 \times 128$, ki ima več kot dva milijona celic. Pri obravnavanju vsake celice program prebere vrednost enajst sosednjih celic, kar lahko privede do veliko zgrešitev v predpomnilniku. Kot smo opisali v razdelku 2.3, lahko med upodobitvijo dveh sosednih sličic za gladko delovanje preteče največ 17 milisekund, korak simulacije sodobnega računalnika na primerno veliki mreži pa je bistveno daljši od tega.

Za rešitev tega problema smo implementirali asinhrono izvajanje simulacije z uporabo knjižnjice GLFW. Program smo konceptualno razdelili na dva modula – prvi nadzoruje simulacijo, drugi pa upodabljanje. Oba modula se

izvajata v svoji niti. Modula nista povsem neodvisna, saj si delita nekatere podatke. Do teh podatkov ne smeta dostopati istočasno.

```
1  void GLFWCALL Simulate( void *arg ) {
2     while( !exit ) {
3        if( !simPaused ) {
4           double startTime = glfwGetTime();
5
6           // Perform the part of the simulation that can be
7           // done asynchronously
8           simulatorModule->stepAsych( simulationData.get() );
9
10          // Lock mutex and do the rest of the simulation
11          glfwLockMutex( simMutex );
12          simulatorModule->stepMutex( simulationData.get(),
13             glfwGetTime() );
14          glfwUnlockMutex( simMutex );
15
16          glfwSleep( 1.0/simulationCap - glfwGetTime() +
17             startTime );
18       }
19    }
20 }
```

Izvorna koda 4.1: Funkcija za asinhrono izvajanje simulacije

Funkcija, ki jo podamo kot parameter funkcije za ustvaritev nove niti, je izpisana v plovki z izvorno kodo 4.1. Pri tej funkciji je kritičen objekt `simulationData`, ki ga uporabljata obe niti. Da preprečimo istočasne dostope smo uporabili razred `GLFWMutex` iz knjižnice GLFW. V programu smo ustvarili instanco tega objekta, poimenovano `simMutex`. Z uporabo metod `glfwLockMutex` in `glfwUnlockMutex` lahko ta objekt "zaklenemo"in "odklenemo". Če kličemo `glfwLockMutex` nad objektom tipa `GLFWMutex`, medtem ko je ta že zaklenjen, bo nit počakala, da se objekt odklene. Z uporabo teh metod in objektov v obeh nitih smo zagotovili varno branje podatkov.

Poglavje 5

Implementacija upodabljanja

V poglavju 4 smo opisali simulator, ki kot rezultat vrne tridimenzionalno polje podatkov. Vsak element polja predstavlja gostoto oblaka za tisti položaj. Izmed metod, opisanih v razdelku 3.2, sta za upodobitev takih volumetričnih podatkov primerni metodi projiciranje vokslov in volumetrično metanje žarkov. Izbrali smo slednjo, saj je pri projiciranju vokslov veliko pisanj v pomnilnik, zaradi česar je ta metoda počasnejša. Pri volumetričnem metanju žarkov imamo tudi več nadzora nad senčenjem in ne potrebujemo zapisov v več medpomnilnikov.

5.1 Uporabljene knjižnice

Pri implementaciji uporabljamo več programskih knjižnic, ki nam omogočajo upodabljanje v uporabniško okno ter programiranje, neodvisno od operacijskega sistema in strojne opreme.

OpenGL (Open Graphics Library) je programski vmesnik za pisanje programov, ki izrisujejo računalniško grafiko. Zagotavlja nivo abstrakcije, ki programerju skrije razlike v strojni opremi. OpenGL podpira več operacijskih sistemov in programskih jezikov.

GLEW (OpenGL Extension Wrangler) je knjižnica, preko katere lahko na

enostaven način kličemo razširitve OpenGL. Potrebujemo jo, ker posamezna grafična kartica tipično podpira le nekatere razširitve.

GLFW (OpenGL Framework) je knjižnica z metodami za odpiranje uporabniškega okna, ustvarjanje OpenGL konteksta in osnovno zaznavanje uporabniškega vnosa. Deluje na več platformah, tako da programerju ni potrebno skrbeti za podrobnosti implementacije.

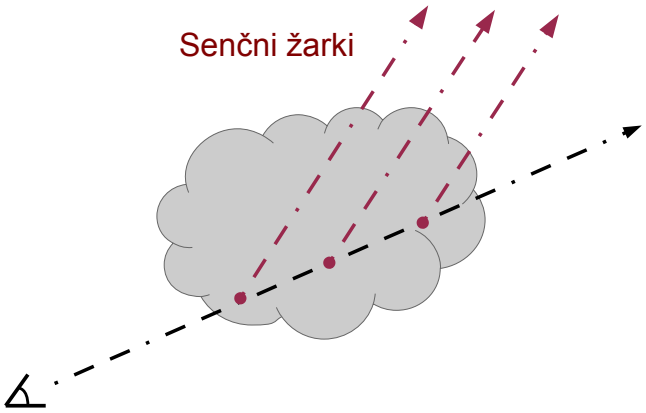
GLM (OpenGL Mathematics) je matematična knjižnica namenjena za računalniško grafiko, zasnovana na programskemu jeziku GLSL. Vsebuje metode za delo z vektorji, matrikami in kvaternioni.

SOIL (Simple OpenGL Image Library) je knjižnica, ki olajša nalaganje tekstur v OpenGL.

5.2 Ideja

Pri senčenju oblakov aproksimiramo enojno sipanje svetlobe in atenuacijo svetlobe pri potovanju skozi oblak. Metoda je zasnovana na volumetričnem metanju žarkov, razširili smo jo pa tako, da pri vsakem vzorcu vržemo senčni žarek (*angl. shadow ray*).

Osnovna ideja senčenja je prikazana na sliki 5.1. Za vsak piksel pošljemo žarek v smer pogleda in ga ekvidistančno vzorčimo. Tam, kjer je vzorčena vrednost višja od nekega prej določenega praga, vemo, da se nahaja oblak. Za te točke hočemo približno izračunati, koliko svetlobe pride do njih. To naredimo tako, da pošljemo senčni žarek v smeri vira svetlobe. Če je virov svetlobe več, lahko za vsak vir svetlobe pošljemo en žarek. Senčni žarek ponovno ekvidistančno vzorčimo. Atenuacijo svetlobe iz dobljenih vzorcev izračunamo tako, da bolj obtežimo tiste, ki se nahajajo bližje izvoru senčnega žarka. Kompozicijo točk na prvotnem žarku izvajamo od najbolj oddaljenega vzorca k najbližjem. Pri tem upoštevamo atenuacijo svetlobe, izračunano iz senčnih žarkov. Vrednost vzorca pa nam pove, v kolikšni meri senčena vrednost te točke prekrije bolj oddaljene točke.



Slika 5.1: Prehod oblaka pri senčenju.

5.3 Senčilnika

Del programa za upodobitev oblakov, ki se izvaja na grafični kartici, smo razvili s programskim jezikom GLSL. Napisali smo senčilnik oglišč (*angl. vertex shader*) in senčilnik fragmentov (*angl. fragment shader*).

5.3.1 Podajanje argumentov

V programu smo definirali kvader, ki vsebuje področje, v kateremu lahko nastanejo oblaki. Da upodobimo oblake, preko OpenGL metod podamo položaj oglišč kvadra in nato kličemo metodo za njegovo upodobitev. Senčilnik tam, kjer ni oblakov, ne upodobi ničesar, tako da celoten kvader nikoli ni viden. Da se izognemo večkratnemu ponavljanju koordinat oglišč, ko jih podajamo kot argumente metodam, jih zapišemo v VBO in ustvarimo ustrezen EBO. Z metodo `glVertexAttribPointer` definiramo razporeditev podatkov v VBO-jih.

Mrežo z gostotami oblakov, ki jo dobimo kot rezultat simulacije, senčilniku podamo preko dinamične 3D teksture. Ker simulacija posodobi vrednosti največ nekajkrat na sekundo, zadnja dva rezultata naprej interpoliramo, kot

je prikazano na plovki z izvorno kodo 5.1. Teksturo ustvarimo z metodo `glTexImage3D`. Ker za vsako polje podamo samo eno vrednost, moramo to izrecno izbrati pri klicu te funkcije z argumentom `GL_RED`.

```
1  void RendererModule::interpolateCloudData(
2     const SimulationData & data, const double time ) {
3
4     int x = data.getGridLength();
5     int y = data.getGridWidth();
6     int z = data.getGridHeight();
7
8     // Calculate relative difference for linear interpolation
9     float relDiff = (time - data.nextTime) /
10       (data.nextTime - data.prevTime);
11    if( relDiff > 1.0f ) relDiff = 1.0f;
12
13    for( int i = 0; i < x; ++i )
14       for( int j = 0; j < y; ++j )
15          for( int k = 0; k < z; ++k )
16             if( data.prevDen[i][j][k] > 0.0f )
17                // Lineary interpolate the density
18                interpolatedData[i][j][k] =
19                   data.prevDen[i][j][k] + relDiff *
20                   ( data.nextDen[i][j][k]
21                   - data.prevDen[i][j][k] );
22             else
23                interpolatedData[i][j][k] = 0.0f;
24 }
```

Izvorna koda 5.1: Interpolacija rezultatov simulacije

Za lažje upravljanje upodabljanja smo implementirali drsnike, s katerimi določimo parametre, kot so robne vrednosti za upodobitev oblakov, izrazitost senc, barva senc in svetlih delov, položaj sonca ter število vzorcev. S spreminjanjem teh vrednosti lahko izrišemo različne oblake (npr. dnevne, večerne).

Nazadnje kličemo metodo `glDrawElements`, da upodobimo oblake. Ker so oblaki polprozorni, je pomembno, da jih upodobimo za objekti, ki ležijo za njimi. Da nam ni potrebno pred vsako naslednjo sličico ponoviti celotnega postopka, smo ustvarili VAO, s katerim lahko hitro preidemo v stanje, ki smo ga dobili z klici metod, opisanih v tem razdelku.

5.3.2 Senčilnik oglišč

Senčilnik oglišč, izpisan v plovki z izvorno kodo A.1, je zelo preprost. Kot parametre sprejme položaj oglišča v prostoru modela `cubeVert` in matriki `view` in `proj`. Prva predstavlja transformacijo iz prostora sveta v prostor pogleda, druga pa je projekcijska matrika, ki tri dimenzionalni prostor projecira na ravnino. Senčilnik vektor pomnoži z matrikama, tako da dobimo položaj oglišča v normaliziranih koordinatah.

5.3.3 Senčilnik fragmentov

V senčilniku fragmentov, izpisani v plovki z izvorno kodo A.2, se izvede poglavitni del upodabljanja oblakov. Senčilnik sprejme argumente preko parametrov tipa `uniform`. Taki argumenti so enaki za vse fragmente, ki jih zajema en OpenGL klic za upodabljanje. Parameter `density` vsebuje 3D teksturo, ki je opisana v razdelku 5.3.1. Sledijo parametri, ki definirajo vidno polje, nato pa več parametrov, ki jih lahko spreminjamo preko uporabniškega vmesnika.

Preden prečkamo 3D teksturo, moramo dobiti vektor, ki je usmerjen od položaja očesa do dela oblaka, ki ga upodabljamo. Preko klica `gl_FragCoord` dobimo koordinate piksla v oknu. Komponenta `x` zavzema vrednost med 0 in širino okna v pikslih, komponenta `y` pa vrednost med 0 in višino okna. Te vrednosti linearno preslikamo v normalizirane koordinate naprave (*angl.*

normalized device coordinates), torej na interval $[-1, 1]$ z enačbama:

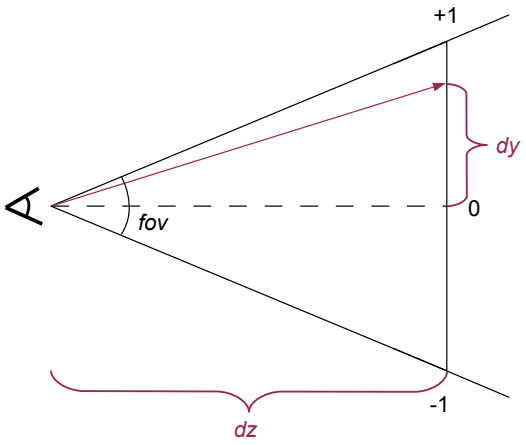
$$dx = \frac{2 \cdot \texttt{gl\_FragCoord.x}}{sw} - 1 \tag{5.1}$$

$$dy = \frac{2 \cdot \texttt{gl\_FragCoord.y}}{sh} - 1, \tag{5.2}$$

kjer je sw širina zaslona v pikslih, sh pa višina zaslona v pikslih. Za izračun tretje komponente vektorja moramo upoštevati širino zornega kota, kakor je prikazano na sliki 5.2. Izračunamo jo z:

$$dz = \tan \frac{fov}{2}, \tag{5.3}$$

kjer je fov širina zornega kota. Dobljeni vektor (dx, dy, dz) nato preslikamo v prostor sveta tako, da ga pomnožimo z inverzom matrike `view`, ki ga je sprejel senčilnik oglišč.



Slika 5.2: Izračun vektorja, usmerjenega v oblak.

Senčilnik ima dve zanki for, s katerima iteriramo preko 3D teksture, kakor smo opisali v razdelku 5.2. V našem programu je edini vir svetlobe sonce, tako da inicializiramo samo en senčni žarek v vsaki iteraciji zunanje zanke. Poleg tega so senčni žarki za vse fragmente enako usmerjeni, saj žarki pri zelo oddaljenih virih svetlobe, kot je sonce, padajo skoraj vzporedno.

Spremenljivka `attenuation` predstavlja delež svetlobe, ki je prispel do vzorčene točke. Inicializiramo jo na vrednost 1 in nato pri vsakemu vzorčenju senčnega žarka pomnožimo z vrednostjo med 0 in 1. Ta vrednost je obratno sorazmerna z vzorčeno vrednostjo, poleg tega je pa obtežena glede na oddaljenost vzorca do izhodišča senčnega žarka.

Ko končamo z senčenjem vzorca ga združimo z do sedaj nabrano vrednostjo z uporabo funkcije `mix`, ki naredi linearno interpolacijo, kjer tretji parameter predstavlja utež – `mix(x,y,a)` vrne vrednost $\mathtt{x}\cdot(1-\mathtt{a})+\mathtt{y}\cdot\mathtt{a}$. Senčilnik kot parameter sprejme tudi barvo senc, ki jo v tem koraku upoštevamo. Utež je sorazmerna z vzorčeno vrednostjo, tako da so bolj gosti deli oblaka manj prozorni.

5.4 Računska zahtevnost in optimizacije

Da lahko oblake upodabljamo z ustreznim številom sličic na sekundo, moramo paziti na zahtevnost računanja. Pri naši implementaciji je ozko žrelo upodabljanja senčenje fragmentov, saj ima njihov senčilnik dve gnezdeni zanki for, v katerih je več računskih operacij, medtem ko je senčilnik oglišč zelo preprost, del upodabljanja, ki se izvaja na CPE, pa tudi relativno nezahteven.

Na CPE izvedemo nekaj računskih operacij, kot so množenje matrik in vektorjev. Najzahtevnejša je računanje inverza matrike, a ker to naredimo le enkrat na sličico, operacija ne zavira izvajanja našega programa. Ker rezultate simulacije pred vsakim izrisom interpoliramo, kakor je prikazano na plovki z izvorno kodo 5.1, moramo iterirati čez tri 3D polja in potencialno za vsak element izvesti operacijo seštevanja, odštevanja in množenja. Tu najprej preverimo ali je prebrana vrednost višja od nič. Če ni, interpolirano vrednost nastavimo na nič, s čimer smo bistveno pohitrili izvajanje.

Potencialno ozko grlo upodabljanja je prepisovanje podatkov iz CPE na GPU. Kadar imamo polje podatkov veliko $128 \times 128 \times 128$, katerega elementi so 32 bitna števila, pri vsakem klicu `glTexImage3D` na GPU prenesemo 8MB

podatkov. Pri 60 sličicah na sekundo prenašamo podatke s hitrostjo 480 MB/s. Starejši PCIe v1.1 ima pasovno širino 4 GB/s [1], kar je nekajkrat več kot potrebujemo. Vseeno se lahko dodaten prenos podatkov pozna, kadar poleg oblakov upodabljamo tudi druge objekte.

Pri senčilniku fragmentov, izpisanem na plovki z izvorno kodo A.2, je pomembno, da čimvečji delež operacij, ki so enake za vse fragmente, opravimo na CPE. Med takimi operacijami je izračun inverza matrike in izračun z komponente vektorja `viewDirection`. Senčilnik ima dve gnezdeni zanki for, zaradi česar je njegovo izvajanje računsko zahtevno. Kolikrat se izvedeta upravljamo s parametroma `viewSamples` in `lightSamples`. Kadar uporabimo manjše vrednosti, pohitrimo izvajanje, vendar je senčenje manj natančno. Notranja zanka se izvede le, kadar je vzorčena vrednost nad nekim pragom. Tudi ta prag lahko upravljamo, s čimer spremenimo računsko zahtevnost in izgled oblaka.

Poglavje 6

Rezultati

Slika 6.1 prikazuje oblake, ki jih upodobi naš program. Za to simulacijo smo uporabili polje z $128 \times 128 \times 128$ celicami, verjetnosti $p_{hum} = 0.1$, $p_{act} = 0.05$ in $p_{cld} = 0.1$ za izračune, opisane z enačbami (4.3) in $e = 22$ za osnovo eksponentne funkcije (4.4). Pri upodabljanju smo naredili 128 vzorcev v zunanji zanki in 64 vzorcev pri senčnih žarkih.

Dodatek B prikazuje zaporedje slik iz simulacije oblakov. Uporabili smo enake parametre, kot pri simulaciji na sliki 6.1. Hitrost upodabljanja je 50 sličic na sekundo na računalniku z procesorjem Core i5 3570k in grafično kartico GeForce GTX 670. Simulacija poteka dovolj hitro, da se opazi premikanje in nastajanje oblakov.

Oblaki so vizualno primerljivi s tistimi iz podobnih volumetričnih metod, kot so na primer opisane v [5, 15]. Take metode imajo v primerjavi z ostalimi nekatere prednosti:

- Omogočajo dinamično simulacijo. Na to simulacijo lahko vplivajo veter, teren in ostale lastnosti sveta, v kateremu se simulacija izvaja.
- Omogočajo dinamično senčenje, ki je odvisno od položaja luči. Uvedemo lahko parametre, s katerimi ima uporabnik nadzor nad izgledom.
- Ne potrebujemo modeliranja.
- Ne potrebujemo sredstev, kot so teksture.

Slika 6.1: Posnetek zaslona pri našem programu.

Vseeno pa imajo volumetrične metode nekatere pomanjkljivosti, kot so:

- Vizualno so manj prepričljive, kot metode, ki uporabljajo slike oblakov.
- Potrebujejo več računske moči.

Zaradi napredka v tehnologiji lahko danes v realnem času upodabljamo podobe, ki so na začetku tisočletja potrebovale več sekund za izris [5]. Kljub temu pa naša metoda še ni primerna za integracijo v večino računalniških iger in simulatorjev. Računalniške igre tipično dodobra izkoristijo strojno opremo, ki je na voljo, tako da uporaba dveh niti za simulacijo, predvsem pa zahtevnega senčenja na grafični kartici, predstavlja preveliko obremenitev za sistem. Menimo, da je pri današnji strojni opremi najbolje integrirati izrisovanje oblakov na podlagi prej upodobljenih (*angl. pre-rendered*) slik, ki jih nato na pameten način dinamično osvetljujemo in premikamo, kot na primer pri metodah opisanih v [8, 16]. Vseeno pa je verjetno, da bodo čez nekaj let vse pogosteje uporabljale metode podobne tej v realno časovnih programih.

Uporabo našega dela v drugih programih bi lahko omogočili na več načinov. Najlažje bi bilo tako, da bi na nekemu koraku stanje simulacije shranili v

datoteko in to jo prebrali drugem programu. Poleg nje bi prebrali in naložili še senčilnik, ter mu poslali podatke iz datoteke in ustrezne parametre, ki odražajo položaj in usmeritev kamere. Slabost takega pristopa je, da se oblaki ne bi premikali. Da bi v program vključili še simulacijo, bi morali dodati še razred `SimulatorModule` in razrede, od katerih je ta razred odvisen. Ker smo program razdelili na več modulov in uporabili objektno orientiran pristop, je večina podatkov enkapsuliranih, tako da bi kaj takega bilo relativno enostavno narediti. Vseeno pa bi lahko za še lažjo integracijo popravili vmesnike do razredov in projekt prevedli v knjižnico.

Poglavje 7

Sklepne ugotovitve

V diplomskem delu smo opisali metode za simulacijo in upodabljanje oblakov. Med sabo se bistveno razlikujejo v prepričljivosti, porabi virov in prilagodljivosti. Podrobneje smo opisali Dobashijevo metodo, ki s celičnim avtomatom učinkovito simulira nastajanje in izginevanje oblakov. Metoda ni povsem točna, saj kompleksno gibanje oblakov le posnema, vendar menimo, da je dovolj prepričljiva za uporabo v nekaterih simulatorjih in računalniških igrah. Pri naši implementaciji smo nekoliko spremenili način izračuna končne teksture, tako da smo z eksponentno funkcijo poostrili robove oblakov. Za upodabljanje smo implementirali metodo, ki tako kot pri volumetričnem metanju žarkov, za vsak piksel končne slike pošlje žarek skozi 3D teksturo. Dodali smo prehod čez senčni žarek, kadar naš prvotni žarek doseže točko, kjer se oblak nahaja, s čimer aproksimiramo enojno sipanje svetlobe. Pri naši implementaciji je potreben le en prehod čez piksle končne slike za izris oblakov.

Razviti metodi za simulacijo in upodabljanje sta dovolj hitri za realnočasovno izvajanje, kadar velikost oblakov in število vzorcev nista prevelika. Omogočata povsem dinamično upodobitev oblakov, ki jo lahko upravljamo z mnogimi parametri, kot so velikost oblakov, položaj virov svetlobe, barvni odtenki oblakov itd. Upodobljeni oblaki niso povsem fotorealistični, vendar je to zaradi njihovega premikanja manj opazno. Poleg tega smo uporabili

take programske knjižnice, da je programska koda prenosljiva. Podobne metode, ki tudi uporabljajo povsem volumetrične oblake, že obstajajo, vendar večina zaradi načina implementacije ali takratnih omejitev strojne opreme ni delovala v realnem času. V računalniških igrah je generiranje in premikanje oblakov le redko implementirano. Kadar igre simulirajo prehod časa z različno osvetlitvijo podnevi in ponoči, so tudi oblaki v njih dinamično osvetljeni, vendar je to senčenje skoraj vedno narejeno na podlagi 2D slik namesto pravih volumetričnih tekstur, kakršne smo mi uporabili.

Naše delo bi lahko na več načinov nadgradili. Pri simulaciji bi lahko poiskali take parametre, oziroma spremenili pravila celičnega avtomata, da bi poleg cumulus oblakov prikazovali tudi ostale vrste oblakov, kot so stratus, cirrocumulus in cumulonimbus. Poleg tega smo pri simulaciji nekoliko preveč potratni s pomnilnikom. Za vsak celični avtomat uporabljamo več tabel, ki so velike toliko, kot celični avtomat. S ponovno uporabo kakšne izmed tabel znotraj koraka simulacije bi lahko število tabel zmanjšali. Problematične so tudi tabele vrednosti tipa `boolean`, saj so te v jeziku C++ veliki en bajt. Velikosti takih tabel bi lahko zmanjšali na eno osmino, če bi uporabili tabele z vrednostmi tipa `char` ali `unsigned char` in bitne operacije, da bi dostopali do ustreznih elementov.

Upodabljanje si lahko prizadevamo narediti bolj vizualno prepričljivo ali manj računsko zahtevno. Najenostavnejši način, da bi izboljšali izgled oblakov, bi bil, da bi poiskali boljše parametre za senčilnik. Kompleksnejša izboljšava bi bila, da bi upoštevali večkratno sipanje svetlobe v oblakih. Navdih za tak pristop lahko najdemo v [2, 3] in starejšemu delu [13]. Našemu programu bi lahko dodali tudi vizualne efekte, kot so žarki svetlobe, kakršne so predstavili v [5] in spreminjanje barve oblaka glede na čas dneva. Za slednje bi lahko uporabili predhodno izračunano teksturo, ki vsebuje barve oblakov glede na čas dneva. S tem bi se izognili zahtevnemu izračunavanju Mievega ali podobnega sipanja. Sečilnik bi lahko optimizirali tako, da bi poiskali minimalno število vzorcev za še ustrezen izgled, kakšen izračun prestavili na CPE in se izognili izračunom, kateri ne vplivajo bistveno na rezultat. Pri

trenutni implementaciji s senčnimi žarki večkrat prečkamo iste dele teksture. Temu bi se lahko izognili tako, da bi na CPE naredili prehod čez teksturo in trenutno akumulirano vrednost shranili v novo teksturo, kakor je opisano v [9, 17], s čimer bi zmanjšali časovno zahtevnost senčenja. Tak pristop je lahko težaven, saj je pisanje v teksturo in sledenje žarkom na CPE počasna operacija, poleg tega bi morali spremeniti način združevanja vzorcev senčnih žarkov.

Dodatek A

Izvorna koda senčilnikov

```
#version 150

in vec3 cubeVert;

uniform mat4 view;
uniform mat4 proj;

void main() {
   gl_Position = proj * view * vec4(cubeVert, 1.0f);
}
```

Izvorna koda A.1: Senčilnik oglišč

```
#version 330

uniform sampler3D density;

// tan( field of view / 2 )
uniform float tanFOV;
// Camera position in world space
uniform vec3 eyePosition;
uniform vec2 screenSize;
uniform mat4 viewInverse;

float near;
float far;

uniform float densityCutoff = 0.06f;
uniform float densityFactor = 0.35f;
uniform float attenuationFactor = 0.05f;
uniform float colorMultiplier = 5.0f;
uniform float lightColorRed = 1.0f;
uniform float lightColorGreen = 1.0f;
uniform float lightColorBlue = 1.0f;
uniform float shadeColorRed = 0.0f;
uniform float shadeColorGreen = 0.0f;
uniform float shadeColorBlue = 0.2f;
uniform float sunPositionX = 0.5f;
uniform float sunPositionY = 0.5f;
uniform float sunPositionZ = 0.5f;

uniform float viewSamplesF = 128.0f;
uniform float lightSamplesF = 64.0f;

int viewSamples = int(viewSamplesF);
int lightSamples = int(lightSamplesF);

const float maxDistance = sqrt(3.0); // Length of a cube
    diagonal
float lightStepSize = maxDistance/viewSamples;
```

```
37
38 out vec4 outColor;
39
40 struct Ray {
41     vec3 origin;
42     vec3 direction; // Normalized
43 };
44
45 // Perform slab method for ray/box intersection. Box spans
       (-1,-1,-1), (1,1,1)
46 // Returns true if there is an intersection
47 // t0, t1 - distances to the two intersections
48 bool IntersectRayBox(Ray r, out float t0, out float t1) {
49     vec3 invR = 1.0 / r.direction;
50     vec3 tbot = invR * (vec3( -1, -1, -1), - r.origin);
51     vec3 ttop = invR * (vec3( 1, 1, 1) - r.origin);
52     vec3 tmin = min(ttop, tbot);
53     vec3 tmax = max(ttop, tbot);
54     vec2 t = max(tmin.xx, tmin.yz);
55     t0 = max(t.x, t.y);
56     t = min(tmax.xx, tmax.yz);
57     t1 = min(t.x, t.y);
58     return t0 <= t1;
59 }
60
61 void main() {
62
63    // Direction in view splace
64    vec3 viewDirection;
65    viewDirection.xy = 2.0f * gl_FragCoord.xy / screenSize -
          1.0f;
66    viewDirection.x *= screenSize.x / screenSize.y;
67    viewDirection.z = -1 / tanFOV;
68
69    // Transform direction to world   space
70    viewDirection = ( viewInverse * vec4( viewDirection, 0 ) )
          .xyz;
71
```

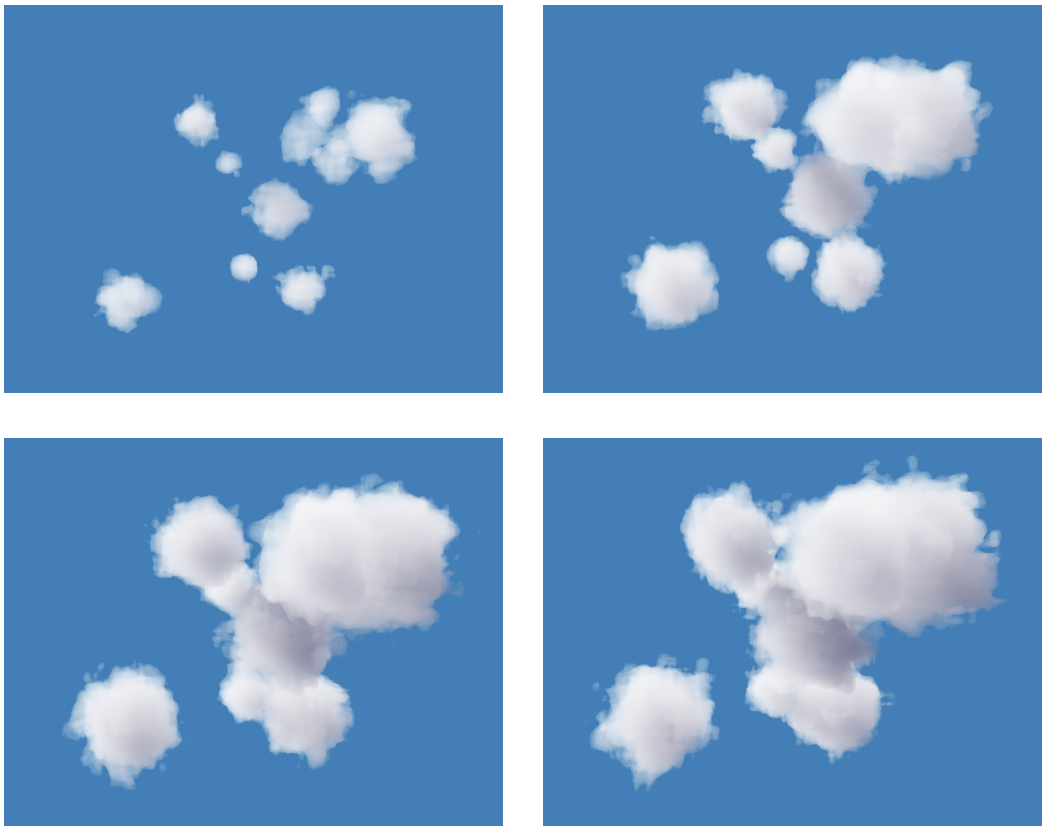
```
72    Ray viewRay = Ray( eyePosition, normalize( viewDirection )
          );
73
74    vec3 color = vec3( 67/256.0, 128/256.0, 183/256.0 );
75    vec3 pos = viewRay.origin;
76    float tmin, tmax;
77    IntersectRayBox( viewRay, tmin, tmax );
78    pos += tmax * viewRay.direction;
79    float viewStepSize = ( tmax - tmin ) / viewSamples;
80
81    vec3 shadeColor = vec3( shadeColorRed, shadeColorGreen,
          shadeColorBlue );
82    vec3 lightColor = vec3( lightColorRed, lightColorGreen,
          lightColorBlue );
83    vec3 sunPosition = vec3( sunPositionX, sunPositionY,
          sunPositionZ );
84
85    for( int i = 0; i < viewSamples; ++i ) {
86
87       float cellDensity = texture( density, pos ).x;
88       if( cellDensity > densityCutoff ) {
89
90          cellDensity *= densityFactor;
91
92          Ray lightRay = Ray( pos, normalize( sunPosition ) );
93
94          float attenuation = 1;
95          vec3 lightPos = pos;
96
97          // Calculate light attenuation
98          for( int j = 0; j < lightSamples; ++j ) {
99             // Closer texture reads contribute more
100            attenuation *= 1 -
101               texture( density, lightPos ).x
102               * attenuationFactor
103               * ( 1 - j / lightSamples );
104            lightPos += lightRay.direction * lightStepSize;
105         }
```

```
106
107         // Add color depending on cell density and
                attenuation
108         if( cellDensity > 0.001 ) {
109            color = mix( color, mix ( shadeColor, lightColor,
                    attenuation ),
110                  cellDensity * colorMultiplier);
111         }
112      }
113
114      pos -= viewRay.direction * viewStepSize;
115
116   }
117
118   outColor = vec4( color, 255 );
119 }
```

Izvorna koda A.2: Senčilnik fragmentov

Dodatek B

Posnetki zaslona



Slika B.1: Evolucija oblakov z našim programom. Med vsakim posnetkom zaslona je preteklo 15 sekund. Manjši oblaki se združijo v večje.

Literatura

[1] T. Akenine-Möller, E. Haines, in N. Hoffman, *Real-Time Rendering 3rd Edition*. Natick, MA, USA: A. K. Peters, Ltd., 2008.

[2] A. Bouthors, "Real-time realistic rendering of clouds," Ph.D. dissertation, Université de Grenoble, 2008.

[3] A. Bouthors, F. Neyret, N. Max, E. Bruneton, in C. Crassin, "Interactive multiple anisotropic scattering in clouds," v *Proceedings of i3D '08*, 2008, str. 173–182.

[4] A. Bressan, "Notes on the boltzmann equation," 2005.

[5] Y. Dobashi, K. Kaneda, H. Yamashita, T. Okita, in T. Nishita, "A simple, efficient method for realistic animation of clouds," v *Procceedings of SIGGRAPH '00*, 2000, str. 19–28.

[6] P. Elinas in W. Stürzlinger, "Real-time rendering of 3d clouds," *Journal on Graphic Tools*, zv. 5, št. 4, str. 33–45, 2000.

[7] K. Englel, M. Hadwiger, J. M. Kniss, in C. Rezk-Salama, *Real-Time Volume Graphics*. Natick, MA, USA: A. K. Peters, Ltd., 2006.

[8] M. J. Harris, "Real-time cloud simulation and rendering," Ph.D. dissertation, University of North Carolina, 2003.

[9] T. Lokovic in E. Veach, "Deep shadow maps," v *Proceedings of SIGGRAPH '00*, 2000, str. 385–392.

[10] J. Monaghan, "Smoothed particle hydrodynamics," *Reports on Progress in Physics*, zv. 68, št. 8, str. 1703–1759, 2005.

[11] M. Mraz, "Mehki celularni avtomati kot koncept za sintezni pristop k modeliranju," *Elektrotehniški vestnik*, zv. 68, št. 2-3, str. 90–96, 2001.

[12] K. Nagel in E. Raschke, "Self-organizing cricality in cloud formation?" *Physica A*, zv. 182, str. 518–531, 1992.

[13] S. Premože, M. Ashikhmon, R. Ramamoorthi, in S. Nayar, "Practical rendering of multiple scattering effects in participating media," v *Proceedings of EGSR '04*, 2004, str. 363–374.

[14] S. Rusinkiewicz in M. Levoy, "Qsplat: A multiresolution point rendering system for large meshes," v *Proceedings of SIGGRAPH '00*, 2000, str. 343–352.

[15] J. Schpok, J. Simons, D. S. Ebert, in C. Hansen, "A real-time cloud modeling, rendering, and animation system," v *Proceedings SIGGRAPH '03*, 2003, str. 160–166.

[16] N. Wang, "Realistic and fast cloud rendering," v *Proceedings of SIGGRAPH '03*, 2003.

[17] C. Yuksel in J. Keyser, "Deep opacity maps," *Computer Graphics Forum*, zv. 27, št. 2, str. 675–680, 2008.