Univerza v Ljubljani

Fakulteta za računalništvo in informatiko

Andrej Bukošek

SISTEM ZA VZPOREDNO IZVAJANJE PODATKOVNO-PRETOČNIH GRAFOV

Diplomsko delo na univerzitetnem študiju

Mentor: doc. dr. Andrej Brodnik
Somentor: prof. dr. Borut Robič

Ljubljana, 2013

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Andrej Bukošek, z vpisno številko 63080072, sem avtor diplomskega dela z naslovom:

SISTEM ZA VZPOREDNO IZVAJANJE PODATKOVNO-PRETOČNIH GRAFOV

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom
  doc. dr. Andreja Brodnika
  in somentorstvom
  prof. dr. Boruta Robiča,
- so elektronska oblika diplomskega dela, naslov (slo., ang.), povzetek (slo., ang.) ter ključne besede (slo., ang.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki „Dela FRI".

V Ljubljani, dne 13. junija 2013 Podpis avtorja:

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: 01908/2013
Datum: 04.03.2013

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

Kandidat: **ANDREJ BUKOŠEK**

Naslov: **SISTEM ZA VZPOREDNO IZVAJANJE PODATKOVNO-PRETOČNIH GRAFOV**

A SYSTEM FOR PARALLEL EXECUTION OF DATA-FLOW GRAPHS

Vrsta naloge: Diplomsko delo univerzitetnega študija

Tematika naloge:

Sodobni računalniški sistemi že dolgo niso več eno procesorski ali eno jedrni. Povečevanje računskih virov, ki so na voljo, zahteva tudi drugačen pristop po eni strani k pisanju programske opreme in po drugi strani k izvajanju le-te.

V diplomskem delu izdelajte sistem, ki bo omogočal vzporedno izvajanje programov, ki bodo modelirani kot aciklični podatkovno-pretočni grafi. V ta namen razvijte jezik, ki bo omogočal definiranje vozlišč grafa in povezav med njimi. Poleg tega razvijte primeren razporejevalnik, ki bo omogočal vzporedno izvajanje tako zapisanih programov. Uporabnost razvitega sistema prikažite na primeru izrisovanje 3D scen in kompozitiranja slik.

Mentor:

doc. dr. Andrej Brodnik

Somentor:

prof. dr. Borut Robič

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

Dekan:

prof. dr. Nikolaj Zimic

Zahvala

Mentorjema, doc. dr. Andreju Brodniku in prof. dr. Borutu Robiču, se zahvaljujem za nasvete, ideje, potrpežljivost in strokovno usmerjanje pri nastajanju diplomskega dela.

Zahvaljujem se tudi vsem, ki ste prispevali predloge in slovnične popravke.

Za podporo, motivacijo in vse ostalo bi se še posebej zahvalil staršem.

Hvala!

Kazalo

Povzetek

Cilj diplomske naloge je bil ustvariti sistem za vzporedno opravljanje poljubnih operacij nad podatki z uporabo podatkovno-pretočnega grafa.

V diplomski nalogi smo razvili sistem za vzporedno izvajanje podatkovno-pretočnih grafov. Vsako vozlišče v grafu izvede neko operacijo nad podatki, ki so vanj vstopili po vhodnih povezavah, rezultate pa pošlje po izhodnih povezavah. Za vzporedno izvajanje operacij smo razvili izvajalnik.

Vsaka operacija je implementirana v dinamično naložljivem modulu ali pa v namenskem programskem jeziku, ki smo ga razvili posebej za to diplomsko nalogo. Za jezik smo razvili in implementirali tudi prevajalnik. Namenski programski jezik je funkcijski in tipno skladen. Sistem tipov v jeziku smo zasnovali modularno. Vsak podatkovni tip je implementiran v zunanjem dinamično naložljivem modulu, ki ga prevajalnik naloži ob svoji inicializaciji. V posameznem modulu so funkcije za generiranje vmesne kode za sistem LLVM, ki na njej izvede optimizacije in jo prevede v strojno kodo.

Kot primer uporabe našega sistema smo razvili operacije za manipulacijo slik, kompozitiranje in izrisovanje 3D scen. Tak nabor operacij se tipično uporablja v filmski industriji, in sicer pri izdelavi posebnih učinkov.

Razvili smo izrisovalnik, ki uporablja algoritem sledenja poti žarkov. Algoritem iz opisa 3D scene ustvari sliko. Postopek temelji na fizikalno korektni simulaciji odbijanja svetlobe po sceni.

Sistem, ki ga sestavljajo prevajalnik, izvajalnik in knjižnica operacij, deluje zadovoljivo in izpolnjuje zastavljene cilje.

Ključne besede: podatkovno-pretočni grafi, povzporejanje, namenski programski jezik, kompozitiranje, izrisovanje, računalniška grafika

Abstract

The goal of this thesis was to create a system for performing arbitrary operations on data in parallel using a data-flow graph.

In this thesis we developed a system for parallel execution of data-flow graphs. Every node in the graph performs an operation, which operates on data passed to it via incoming graph edges and sends results over outgoing edges. We developed an execution engine for executing operations in parallel.

Each operation is implemented in a dynamically loadable module or using a domain-specific language, which was developed specifically for this thesis. We also developed and implemented a compiler for this language. The domain-specific language is functional and strongly typed. We designed its type system to be modular. Every data type is implemented in an external dynamically loadable module, which the compiler loads during its initialization. Each module contains functions for generating an intermediate representation for the LLVM system, which optimizes it and translates it into machine code.

As an example usage of our system we developed operations for image manipulation, compositing, and rendering of 3D scenes. Such a set of operations is commonly used in the film industry for the creation of special effects.

We developed a renderer based on the path tracing algorithm, which creates an image from the description of a 3D scene. This method is based on a physically-correct simulation of light bouncing around the scene.

The system, which consists of a compiler, execution engine, and a library of operations, is functioning satisfactory and fulfills the set goals.

Keywords: data-flow graphs, parallelization, domain-specific language, compositing, rendering, computer graphics

POGLAVJE 1

Uvod

Pri izdelavi filmov postajajo računalniško generirani vizualni učinki vedno bolj nepogrešljivi. Dve ključni kategoriji vizualnih učinkov sta kompozitiranje in animacija.

Kompozitiranje je proces kombiniranja vizualnih elementov iz različnih virov v enotno celoto, pri čemer je cilj ustvariti vtis, da so vsi elementi del iste scene. Tipičen primer kompozitiranja je zamenjava zelenega ali modrega platna (slika 1.1a) v sceni z računalniško generirano sliko ali animacijo nekega namišljenega prostora (slika 1.1b).

(a) Originalni posnetek.

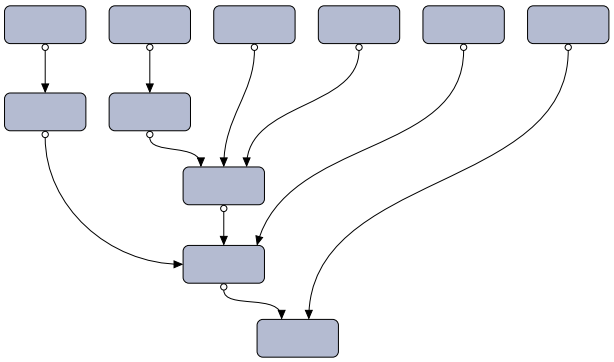
(b) Isti posnetek po obdelavi.

Slika 1.1: Kompozitiranje pri filmu Hobit (vir: [6]).

Animacija vključuje modeliranje, teksturiranje, animiranje in izrisovanje računalniško generiranih 3D likov, ozadij, delcev in lokacij.

Trenutno najboljše orodje, ki združuje obe področji in omogoča ustvarjanje kompleksnih vizualnih učinkov, je Houdini [28]. Glavna težava tega programa je, da je zelo drag in ima omejeno razširljivost, saj ni odprtokoden. Poleg tega ima zelo zapleten vmesnik in se ga je zato težko naučiti uporabljati.

Glavni cilj te diplomske naloge je ustvariti posplošeno odprtokodno verzijo takega orodja, ki bi bila sposobna opravljati poljubne operacije nad podatki.



Slika 1.2: Primer strukture grafa.

Temeljna funkcionalnost programa je vzporedno izvajanje podatkovno-pretočnih (ang. *data-flow*) grafov (slika 1.2). Vsako vozlišče v grafu predstavlja neko operacijo nad podatki, ki se prenašajo po povezavah med vozlišči.

Za implementacijo operacij v grafu je bil razvit tudi namenski programski jezik, katerega cilj je čimbolj poenostaviti razvoj novih operacij ljudem, ki še ne znajo programirati. Operacije so lahko implementirane tudi v kateremkoli drugem jeziku, ki podpira prevajanje kode v dinamično naložljiv modul (ang. *shared object*).

Kot primer uporabe sistema so bile razvite operacije, ki omogočajo enostavno kompoziti-ranje in 3D animacijo. Med drugim je bil implementiran tudi zelo napreden izrisovalnik 3D scen, ki uporablja algoritem sledenja poti žarkov (ang. *path tracing*), za katerega velja, da ustvarja najbolj realistične (fizikalno korektne) slike.

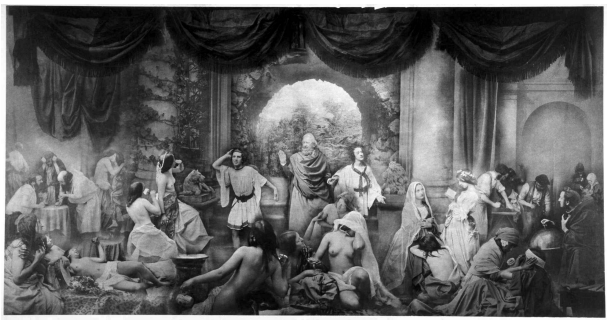
V preostanku tega poglavja je na kratko predstavljena zgodovina in uporaba kompoziti-ranja in animacije. Drugo poglavje opisuje arhitekturo sistema, torej, katere komponente vsebuje in kako so le-te med seboj povezane. V tretjem poglavju sta opisana programski jezik in prevajalnik zanj, oba sta bila razvita za potrebe te diplomske naloge. Četrto poglavje vsebuje opis sistema za izvajanje podatkovno-pretočnih grafov in razporejevalnika. V petem poglavju je opisanih nekaj področij uporabe sistema ter operacije, ki so bile razvite za posamezno področje. V zaključku je kratek povzetek opravljenega dela ter ideje za izboljšave celotnega sistema in morebitno nadaljnje delo.

1.1 Uvod v kompozitiranje

Glavni namen kompozitiranja je združitev ločenih elementov (element je lahko posamezna slika, animacija ali kaj podobnega) v enotno celoto, pri čemer je zelo pomembno, da končni kompozit izgleda tako, kot da bi ga v celoti posnela ista kamera — da na prvi pogled ni očitno, da je bilo sploh uporabljeno kompozitiranje [4].

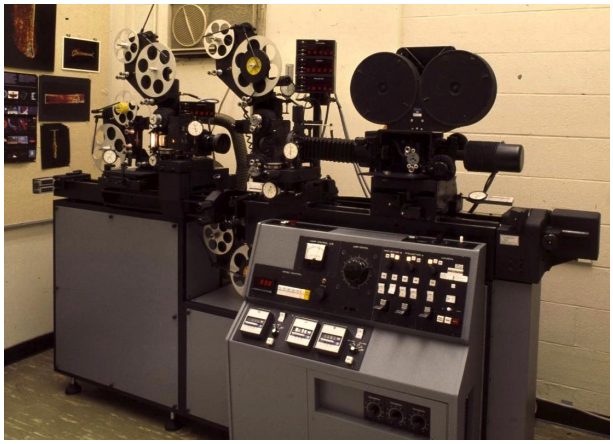
Takorekoč vsi ljudje so že videli rezultate kompozitiranja, četudi ne gledajo filmov. Poleg posebnih učinkov v filmih je daleč najbolj razširjena uporaba kompozitiranja pri vremenski napovedi na televiziji in celo pri reklamah. Vremenar kaže na platno zelene ali modre barve, računalnik pa zamenja platno s sliko države in animacijo vremenske napovedi v posameznih krajih. Zamenjava temelji na barvi, zato vremenarji nikoli ne nosijo zelenih oz. modrih oblačil — če bi jih, bi na končnem kompozitu ti kosi oblačil izgledali prosojno.

Začetki kompozitiranja segajo že v leto 1857, ko je švedski fotograf Oscar Gustave Rejlander združil 32 različnih negativov v eno samo ogromno sliko (slika 1.3). Če bi to sliko želel ustvariti z enim samim negativom, bi rabil ogromen studio in veliko ljudi. Z uporabo kompozitiranja pa je posnel ločene manjše skupine ljudi in jih kasneje združil skupaj [4].



Slika 1.3: Oscar Gustave Rejlander, *Dva načina življenja*, 1857 (vir: [24]).

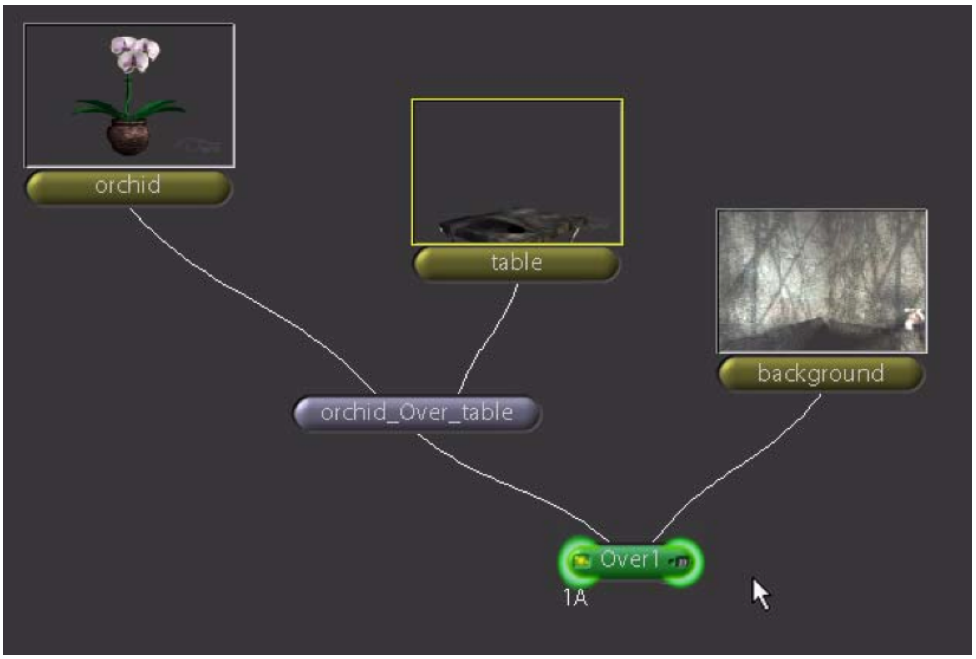
Konec 19. stoletja se je pojavila filmska industrija, kot jo poznamo danes. Seveda so tudi v filmih želeli uporabiti tehnike kompozitiranja, za kar so razvili optične tiskalnike (slika 1.4), na katerih so lahko opravljali optično kompozitiranje. Le-to je dandanes sicer zastarelo, vendar so kljub temu osnovni koncepti skoraj enaki kot pri modernem digitalnem kompozitiranju.



Slika 1.4: Optični tiskalnik (vir: [11]).

Kompozitiranje je v osnovi maskiranje delov slik in kombiniranje takih slik na različne načine. Velik del tega predstavlja tudi manipulacija slik, npr. korekcija barv, kontrasta, ustvarjanje maske na podlagi barve ipd.

Za lažje razumevanje si poglejmo primer preprostega kompozitiranja z grafi iz dokumentacije programa Shake [1]:



Slika 1.5: Preprost primer kompozitiranja, graf (vir: [1]).

Graf na sliki 1.5 ustvari sledeči kompozit:

Slika 1.6: Preprost primer kompozitiranja, končni kompozit (vir: [1]).

Na sliki 1.6 je prikazan rezultat kompozitiranja treh slik — orhideje, mize in ozadja. V praksi se slike ne bi ujemale tako lepo, ampak bi bilo potrebno kakšno zamakniti ter popraviti barve in sence. Za implementirane operacije s tega področja glej podpoglavje Kompozitiranje na strani 34.

1.2 Uvod v animacijo

Animacija je ustvarjanje sekvenc slik, ki ob predvajanju dajejo vtis gibanja.

Začetki moderne animacije segajo v leto 180. Takrat je Ting Huan na Kitajskem izumil prvi zoetrop (slika 1.7). Na notranji strani cilindra so posamezne slike iz sekvence, cilinder pa ima navpične zareze, skozi katere lahko opazovalec gleda animacijo, ki nastane med vrtenjem cilindra. Brez zarez bi se slike med vrtenjem zabrisale, tako pa opazovalec vidi hitro menjajoče se slike, kar možgani prevedejo v gibanje.

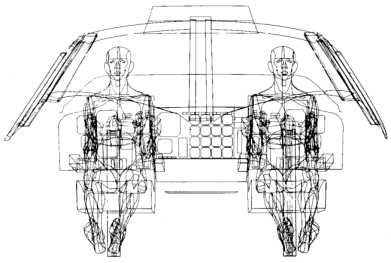
Slika 1.7: Zoetrop (vir: [34]).

Z razvojem filmske industrije konec 19. stoletja je postala animacija dosti bolj dostopna. Na začetku so delali animacije tako, da so za vsako spremembo ustavili kamero, popravili sceno in potem nadaljevali s snemanjem (modernejši primer takega načina animacije je serija čeških risank *A je to!*). Kasneje so risali posamezne sličice animacije na papir in jih prenesli na negativ. Izboljšava te tehnike je t.i. *cel* animacija, kjer so slike s papirja prerisali ali fotokopirali na folije celuloidnega filma (od tu ime *cel*) in jih ročno pobarvali, na koncu so celuloiden film presneli na film za predvajanje v kinodvoranah.

Danes se uporablja izključno računalniška animacija. V to področje se v kontekstu filmske industrije uvršča tudi modeliranje (ustvarjanje tridimenzionalnih modelov), teksturiranje („poslikava“ modelov, določanje materialov in površine) in izrisovanje (pretvorba scene v sliko).

Prvi poskusi uporabe računalnikov za ustvarjanje animacij so bili opravljeni že v času 2. svetovne vojne [26]. John Whitney je povezal servo motorje na analogne računalnike, ki so nadzorovali gibanje luči in predmetov. Najbolj znana uporaba tega pristopa je v filmu Stanleya Kubricka *2001: A Space Odyssey* iz leta 1968.

Prva 3D animacija je nastala leta 1964 v Lawrence Livermore National Laboratory. Predstavljala je žične modele ljudi (slika 1.8), nastala pa je v okviru razvoja ergonomskega opisa človeškega telesa za letalsko podjetje Boeing.



Slika 1.8: William Fetter, *Boeing man*, 1964 (vir: [7]).

V filmski industriji so animirane 3D žične modele uporabili šele v drugi polovici 70. let 20. stoletja. Najbolj znana uporaba te tehnologije je v filmu *Star Wars* iz leta 1977.

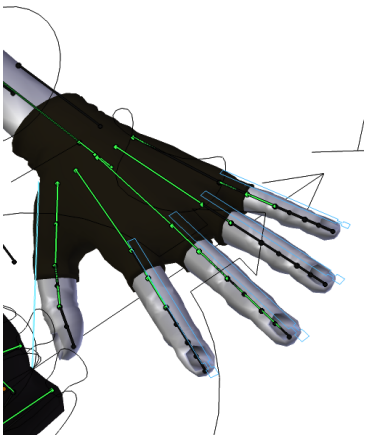
V 80. letih so računalniki postali hitrejši, pojavili pa so se tudi namenski računalniki za delo s 3D grafiko (podjetje *Silicon Graphics, Inc.* je dolgo

časa izdelovalo tehnično najbolj napredne sisteme s tega področja). Ti so omogočali interaktivno izrisovanje 3D scen, kar je olajšalo ustvarjanje animacij.

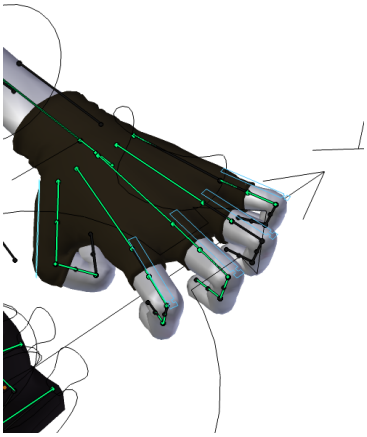
Prva uporaba zapolnjenih (torej, ne žičnih) 3D modelov v večji filmski produkciji je bila v Disneyevem filmu *Tron* iz leta 1982. Ironično je, da film ni bil nominiran za oskarja za posebne učinke, saj je akademija bila mnenja, da so z uporabo računalnika goljufali.

V filmu *The Abyss* iz leta 1989 so prvič združili fotorealistične računalniško generirane posnetke s pravimi. V začetku 90. let je bila tehnologija že dovolj razvita, da so jo lahko uporabljali v vedno več filmih. Dve najbolj vidni uporabi le-te sta v filmih *Terminator 2* (1991) in *Jurassic Park* (1993).

Leta 1995 je Pixar izdal film *Toy Story*, ki je bil prvi daljši popolnoma računalniško animiran film.



(a) Okostje modela roke.



(b) Model po premiku okostja.

Slika 1.9: Blender Foundation, *Prikaz okostja roke iz projekta* Sintel, 2011 (vir: [8]).

V grobem je postopek izdelave 3D animacije sledeč:

- naredimo tridimenzionalne modele — temu pravimo modeliranje,
- modelom dodamo okostje — le-to je sestavljeno iz rigidnih kosti, povezanimi s sklepi, in določa, kako se lahko deli modela premikajo v povezavi z ostalimi, podobno kot pri človeškem okostju (glej sliko 1.9a),
- določimo ključne točke animacije — za vsako točko okostje premaknemo v takšen položaj, kot ga želimo (glej sliko 1.9b); ključne točke so po navadi več sličic narazen, vmesne sličice pa interpolira računalnik,
- izrišemo animacijo.

Interpolacijo vmesnih premikov lahko nadzorujemo s krivuljami s kontrolnimi točkami (npr. z Bézierjevimi krivuljami).

Postopek animiranja naravnega gibanja je zelo zahteven, zato so izumili sisteme za zajem gibanja. Sistemov je več vrst, večina jih temelji na zaznavi gibanja posebnih markerjev. Le-te pritrdimo na osebek ali predmet, katerega gibanje želimo posneti, potem pa sistem preko senzorjev (ti so lahko navadne ali infrardeče kamere, magnetni senzorji ali kaj drugega) zajame položaje markerjev skozi čas. Ko imamo posnetek gibanja, ga lahko naložimo v program za animacijo, kjer določimo korespondenco točk med dejanskim in namišljenim likom. Za implementirane operacije s tega področja glej podpoglavje Izrisovanje na strani 41.

POGLAVJE 2

Arhitektura sistema

V tem poglavju se nahaja opis arhitekture sistema in njegovega delovanja.

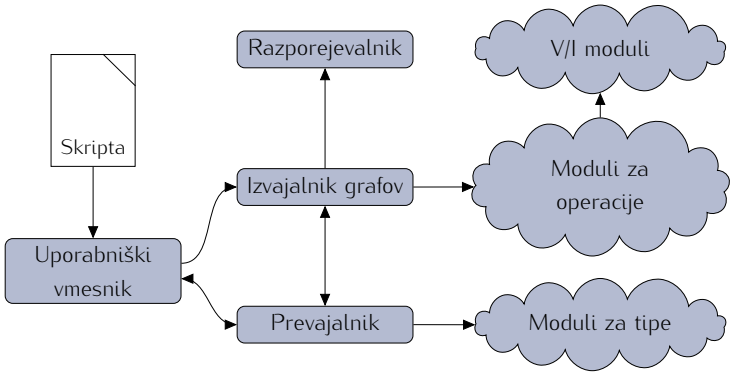
2.1 Opis arhitekture

Sistem, razvit v diplomski nalogi, ima tri glavne sklope:

- sistem za vzporedno **izvajanje** podatkovno-pretočnih (ang. *data-flow*) grafov,
- funkcijski programski **jezik**,
- implementacija **operacij** za kompozitiranje in izrisovanje 3D scen.

Vse to je združeno v en sam program. Komponente tega programa so (slika 2.1):

- razporejevalnik opravil,
- izvajalnik grafov,
- zunanji moduli za operacije,
- prevajalnik za namenski jezik,
- zunanji moduli za podatkovne tipe jezika,



Slika 2.1: Arhitektura sistema.

- zunanji vhodno-izhodni moduli (za branje in pisanje slik in filmov ter dostopa do strojnih senzorjev),
- uporabniški vmesnik (ukazni oz. grafični; inicializira ostale komponente in izvede skripto s programom oz. nudi okolje za izdelavo le-te).

2.2 Delovanje sistema

V grobem je potek delovanja sistema sledeč; sistem:

- sprejme skripto, napisano v namenskem programskem jeziku, opisanem v poglavju Jezik na strani 9,
- naloži vhodno-izhodne module iz mape `ios`,
- naloži module za operacije iz mape `ops`,
- inicializira prevajalnik, ki naloži module za tipe iz mape `types`,
- v jezik vstavi konstruktorje za operacije, kot je opisano v poglavju Izvajanje na strani 25,
- s prevajalnikom prevede skripto,
- se premakne v mapo, v kateri se nahaja skripta,
- inicializira razporejevalnik opravil,
- izvede prevedeno skripto iz pomnilnika (pokliče funkcijo `main()`),
- po izvajanju skripte izvede generiran graf in počaka, da se izvajanje grafa zaključi.

Podrobnejše delovanje posameznih korakov je opisano v poglavjih, ki sledijo.

POGLAVJE 3

Jezik

Namen programskega jezika, razvitega v tej diplomski nalogi, je povezovanje komponent sistema med seboj. Jezik se uporablja za ustvarjanje grafov preko klica funkcij za ustvarjanje vozlišč in povezovanje le-teh. V jeziku so lahko implementirane tudi razne operacije (vozlišča) za graf. To uporabnikom poenostavi ustvarjanje novih operacij.

3.1 Uvod v programske jezike

Programski jeziki so umetni jeziki, namenjeni natančnemu opisu algoritmov v obliki, ki jo računalnik lahko razume.

Zgodovina. Prvi programski jezik je nastal za Jacquardove statve, izumljene leta 1801. Te so uporabljale luknjane kartice, na katerih so luknje opisovale premike kavljev za dvigovanje in spuščanje niti. Na ta način se je dalo opisati postopek (algoritem) za izdelavo tkanin s kompleksnimi vzorci [29]. Luknjane kartice so za vnos podatkov in programov uporabljali tudi prvi računalniki.

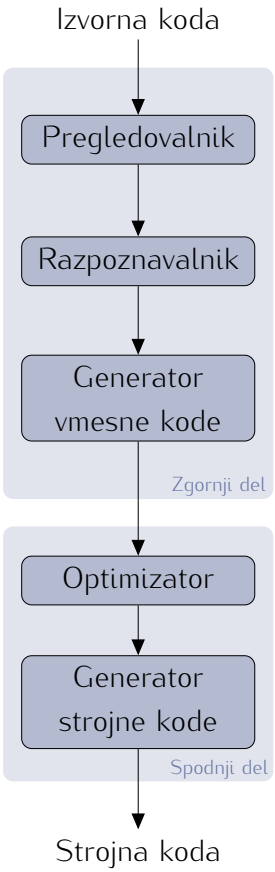
V 30. in 40. letih 20. stoletja so bili razviti formalizmi za predstavitev algoritmov na matematičen način (Churchev lambda račun, Turingovi stroji) [27].

Programiranje prvih računalnikov je potekalo kar prek vnašanja strojne kode v računalnik. Kasneje se je pojavil zbirni jezik (ang. *assembly*), ki je olajšal to opravilo — posamezni

ukazi so bili predstavljeni v človeško lažje berljivi obliki, ki jo je program prevedel v strojno kodo.

Prvi moderni programski jezik je bil FORTRAN (*FORmula TRANslator*, 1955) [27]. Ta je višjenivojske konstrukte (aritmetične izraze, zanke, funkcije) prevedel v zbirni jezik, ki se je potem prevedel v strojno kodo. FORTRAN je imel prvi popolnoma delujoč prevajalnik leta 1957.

Prevajalnik. Osnovni namen prevajalnika je, da pretvori izvorno kodo, napisano v nekem programskem jeziku, v strojno kodo [35].



Slika 3.1: Arhitektura prevajalnika.

Na sliki 3.1 je prikazana arhitektura tipičnega prevajalnika.

Pregledovalnik (ang. *lexer*) ugotavlja, katere besede jezika so v izvorni kodi (t.j. ali gre za številko, niz, rezervirano besedo, ime simbola itd.).

Razpoznavalnik (ang. *parser*) iz zaporedja besed jezika gradi sintaksno drevo, ki ustreza formalni gramatiki jezika. Zatem se na tem drevesu izvede semantična analiza, ki poveže reference spremenljivk in funkcij na njihove definicije ter preveri ujemanje podatkovnih tipov. V tem koraku prevajalnik tudi ugotovi, ali je program sintaktično in semantično pravilen; morebitne napake javi uporabniku.

Generator vmesne kode pretvori sintaksno drevo v vmesno obliko, ki je že zelo podobna procesorskim ukazom, vendar je neodvisna od specifičnega procesorja in lahko vsebuje dodatne semantične podatke, kar poenostavi nadaljnje pretvorbe programa.

Optimizator izvaja transformacije na vmesni kodi. Le-te lahko pohitrijo izvajanje, zmanjšajo velikost generirane kode in celo zmanjšajo energijske zahteve procesorja med izvajanjem [10].

Končni korak prevajanja je generiranje strojne kode iz vmesne kode. To strojno kodo se lahko spravi v datoteko ali pa izvede kar v pomnilniku.

Oblika BNF. Formalno gramatiko jezika se lahko zapiše v obliki BNF (Backus-Naur Form) [3].

Gramatika v BNF sestoji iz pravil oblike: ⟨*simbol*⟩ ::= izraz.

⟨*simbol*⟩ predstavlja sintaktično spremenljivko, ki se lahko pojavi v izrazih. Izraz sestoji iz enega ali več zaporedij simbolov. V primeru več zaporedij le-te ločimo z znakom |, ki si ga lahko predstavljamo kot besedo „ali". Simboli, ki se nikoli ne pojavijo na levi strani pravila, so terminali (končni simboli). ::= pomeni, da se simbol na levi lahko zamenja z izrazom na desni.

Opcijske izraze se lahko da med oglate oklepaje. Če se nek izraz ponovi 0 ali večkrat, ga lahko damo med zavite oklepaje oz. na konec damo zvezdico, za vsaj eno ponovitev pa na konec damo plus. Podizraze lahko z navadnimi oklepaji združimo v skupine. Nize damo med navednice.

Za lažje razumevanje si poglejmo primer gramatike za preproste aritmetične izraze:

⟨*izraz*⟩ ::= ⟨*clen*⟩ | ⟨*izraz*⟩ '+' ⟨*clen*⟩

⟨*clen*⟩ ::= ⟨*produkt*⟩ | ⟨*clen*⟩ '∗' ⟨*produkt*⟩

⟨*produkt*⟩ ::= ⟨*stevilka*⟩ | '(' ⟨*izraz*⟩ ')'

⟨*stevilka*⟩ ::= ⟨*cifra*⟩+

⟨*cifra*⟩ ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

Pogosto se osnovne elemente (kot je ⟨*cifra*⟩ v zgornjem primeru) zaradi preglednosti izpusti, oz. omeni v spremnem besedilu.

Vrste gramatik. BNF lahko opisuje zgolj kontekstno neodvisne gramatike (KNG), vendar so te zadosti močne za opis vseh praktičnih programskih jezikov.

Kontekstno neodvisne gramatike lahko razdelimo na nekaj podrazredov:

- $LR(k)$ gramatike — opisujejo deterministične KNG in se jih zato lažje razpoznava (parameter k pove, koliko besed naprej mora razpoznavalnik gledati, da lahko uspešno razpozna jezik te gramatike),
- $LL(k)$ gramatike — so še bolj omejene od $LR(k)$, pri razpoznavanju ne sme priti do vračanja (ang. *backtracking*),
- linearne gramatike — nimajo pravil z več kot enim neterminalom na desni strani,
- regularne gramatike — najbolj omejene, opisujejo regularne jezike (te se razpoznava s končnimi avtomati).

Najbolj popularne so *LL*(1) gramatike, saj se jih zelo enostavno (in hitro) razpoznava. Razpoznavalnik za tako gramatiko se lahko napiše tudi ročno (razpoznavalnik z rekurzivnim spustom, za opis glej podpoglavje Prevajalnik na strani 18), medtem ko pri ostalih gramatikah po navadi uporabljamo avtomatsko generirane razpoznavalnike (npr. z orodjem *yacc*).

Paradigme jezikov. Programske jezike lahko razdelimo glede na paradigmo programiranja. Paradigma določa model oz. način programiranja, ki je skupen vsem jezikom z določeno paradigmo.

Glede na glavne štiri paradigme so jeziki lahko [32]:

- imperativni (program je opisan kot zaporedje ukazov oz. korakov, ki vodijo do rešitve),
- funkcijski (program je opisan s kompozitumom funkcij; v takih jezikih je rekurzija bolj naravna od iteracije),
- objektno usmerjeni (vsak koncept v programu je objekt z metodami),
- deklarativni (opisuje logiko programa, ne pa korakov, s katerimi se pride do rešitve).

V praksi jeziki ne spadajo zgolj v eno paradigmo.

Obravnava podatkovnih tipov. Programski jeziki obravnavajo podatkovne tipe na različne načine. V splošnem se delijo na tipno skladne in neskladne.

V tipno neskladnih jezikih je pretvorba med tipi implicitna, vendar ni vedno taka, kot bi si želeli. Na primer: `a = "4" + 2` — pri šibko tipiranih jezikih lahko prevajalnik pretvori `"4"` v `4` ali pa `2` v `"2"`, torej dobimo rezultata `6` ali `"42"`. Pri takih jezikih lahko po navadi v isto spremenljivko enkrat damo celo število, drugič pa niz znakov.

V tipno skladnih jezikih prevajalnik ne dela implicitnih pretvorb med tipi. Vsako pretvorbo moramo izvesti ročno. Tak pristop je mnogo boljši, saj ujame pogoste programerske napake, poleg tega pa omogoča prevajalniku, da bolje optimizira kodo (v posamezno spremenljivko lahko damo samo vrednosti dotičnega tipa).

3.2 Uvod v LLVM

LLVM[1] [16] je C++ knjižnica, ki nudi spodnji del infrastrukture prevajalnika — implementira optimizator in generator strojne kode (glej sliko 3.1 na strani 10).

Kot vhod sprejme vmesno kodo (IR, ang. *intermediate representation*), ki je podobna zbirnemu jeziku za kakšen RISC procesor, le da ima neskončno registrov in podporo za poljubne tipe (tudi strukture). Pogosto rabljene osnovne tipe ima že definirane (npr. 64-bitna cela števila). LLVM tipe obravnava strogo, kar pomeni, da v register, ki ima trenutno celoštevilsko vrednost, ne moremo spraviti realnega števila brez vmesne pretvorbe. Dva tipa z isto predstavitvijo v pomnilniku se smatrata za enaka, tudi če jima določimo različni imeni.

IR lahko gradimo neposredno v jeziku C++ (z metodami razreda `IRBuilder`) ali pa naložimo iz datoteke. V datoteki je vmesna koda lahko predstavljena v tekstovni obliki (sintaksa je podobna zbirnemu jeziku) ali pa v binarni obliki. Pri izdelavi novih prevajalnikov se spodbuja neposredno grajenje vmesne kode.

Na podani IR lahko LLVM izvede veliko naprednih optimizacij (za primer glej podpoglavje Prevajalnik na strani 18).

LLVM podpira veliko arhitektur (x86, MIPS, PowerPC, ARM, Sparc ...) in lahko iz IR generira strojno kodo v glavnem pomnilniku, tako da jo je mogoče izvesti brez pisanja v vmesne datoteke.

3.3 Jezik

Jezik, razvit v tej diplomski nalogi, je funkcijski, strogo tipiran in preveden v strojno kodo. Za optimizacije in generiranje strojne kode je uporabljen sistem LLVM, opisan v podpoglavju Uvod v LLVM zgoraj.

Sistem tipov. Tipi v jeziku so implementirani preko dinamično naložljivih modulov. Vsak modul implementira funkcije za generiranje kode za operatorje, ki so smiselni za dotični podatkovni tip (npr. seštevanje dveh celih števil, množenje matrike z vektorjem ipd.). Moduli za tipe morajo biti implementirani v jeziku C++, saj za generiranje kode uporabljajo funkcije iz knjižnice LLVM.

Poleg operatorjev lahko posamezen modul generira kodo za knjižnico funkcij za nek tip (npr. pri vektorjih so v knjižnici funkcije za računanje skalarnega in vektorskega produkta). Pri sestavljenih tipih (kot so na primer vektorji in matrike) je ena od funkcij tudi konstruktor

[1] Včasih je kratica LLVM pomenila *Low Level Virtual Machine*, ampak so jo opustili. Polno ime projekta je sedaj samo LLVM.

za ta tip. Konstruktor pridobi pomnilnik za strukturo in nastavi vrednost elementov na parametre konstruktorja.

Za lažji razvoj tipov so bili implementirani makri. Ti zmanjšajo količino kode, ki je potrebna za deklaracijo novega tipa, implementacijo posameznih operatorjev, dodajanje funkcij v knjižnico itd. Implementacija makrov je v datoteki `types.h`.

Na začetku vsakega modula je potrebno definirati, za kateri tip gre in kako se ta tip predstavi v LLVM sistemu. Poglejmo si prvi dve vrstici kode iz modula za cela števila:

```
1 #include "types.h"
2
3 DECLARE_TYPE("Integer", "int", llvm::Type::getInt64Ty(ctx))
```

V tem primeru je `"Integer"` polno ime tipa, `"int"` je ključna beseda (ang. *keyword*) tega tipa (le-to se uporablja pri pisanju programov v tem programskem jeziku), `llvm::Type::getInt64Ty(ctx)` pa določa 64-bitni celoštevilski tip v LLVM obliki.

Spodnja koda prikazuje implementacijo operatorja za seštevanje dveh 3D vektorjev:

```
1  DECLARE_OP(add, vec3, vec3)
2  {
3      /* Load vector components */
4      Value *v1x = IR.CreateLoad(IR.CreateStructGEP(L, 0));
5      Value *v1y = IR.CreateLoad(IR.CreateStructGEP(L, 1));
6      Value *v1z = IR.CreateLoad(IR.CreateStructGEP(L, 2));
7
8      Value *v2x = IR.CreateLoad(IR.CreateStructGEP(R, 0));
9      Value *v2y = IR.CreateLoad(IR.CreateStructGEP(R, 1));
10     Value *v2z = IR.CreateLoad(IR.CreateStructGEP(R, 2));
11
12     /* Allocate memory and compute result */
13     std::vector<Value *> vec3_args(3);
14
15     vec3_args[0] = IR.CreateFAdd(v1x, v2x, "vaddtmpx");
16     vec3_args[1] = IR.CreateFAdd(v1y, v2y, "vaddtmpy");
17     vec3_args[2] = IR.CreateFAdd(v1z, v2z, "vaddtmpz");
18
19     return IR.CreateCall(_mod->getFunction("vec3"), vec3_args, "vec3tmp");
20 }
```

Koda najprej naloži posamezne komponente, jih sešteje in jih poda kot parametre konstruktorju za 3D vektor.

Pri skalarnih tipih je ta operator dosti preprostejši:

```
1 DECLARE_OP(add, int, int) {
2     return IR.CreateAdd(L, R, "iaddtmp");
3 }
```

Velika prednost generiranja kode za operatorje in knjižnične funkcije je v tem, da omogoča nadaljne optimizacije. Če bi moduli implementirali zgolj funkcije, ki bi jih koda klicala, ne da bi vedela, iz kakšnih ukazov so sestavljene, bi bila hitrost izvajanja dosti manjša.

Poleg tega lahko vsak tip uporabi poljubno sintakso za konstante (pregledovalnik med branjem konstant preda popoln nadzor modulom za tipe). Sintaksa konstant je tako popolnoma poljubna, kar poveča fleksibilnost.

Tip	Opis	Konstante
`bool`	Boolov tip	`true|false`
`int`	64-bitni celoštevilski tip	`[0-9]+`
`mat3`	3×3 matrika	brez
`mat4`	4×4 matrika	brez
`real`	64-bitno realno število	`[0-9]+\.[0-9]+`
`string`	niz znakov	`"[^"]*"`
`vec3`	3D vektor	brez
`brdf`	model odboja svetlobe od materiala	brez
`camera`	parametri kamere	brez
`color`	RGBA barva	brez
`image`	slika	brez
`material`	opis materiala za izrisovalnik	brez
`node`	vozlišče v grafu operacij	brez
`obj`	objekt za izrisovalnik	brez
`sgnode`	vozlišče v hierarhiji objektov scene	brez
`spectrum`	odbojnost glede na valovne dolžine (barva materiala)	brez

Tabela 3.1: Trenutno implementirani tipi.

V tabeli 3.1 so predstavljeni trenutno implementirani podatkovni tipi. Pri tipih, ki nimajo konstant, je mišljeno, da uporabnik kliče konstruktor za dotični tip. Optimizator bo ugotovil, da gre za konstanto in jo primerno obravnaval.

Gramatika jezika. Ciljni uporabniki celotnega sistema vključujejo tudi ljudi, ki še ne znajo programirati, kar je bilo upoštevano pri zasnovi jezika.

Gramatika jezika je sledeča:

⟨*identifierexpr*⟩ ::= ⟨*identifier*⟩ | ⟨*identifier*⟩ '`(`' ⟨*expression*⟩* '`)`'

⟨*constexpr*⟩ ::= ⟨*const*⟩

⟨*parenexpr*⟩ ::= '`(`' ⟨*expression*⟩ '`)`'

⟨*ifexpr*⟩ ::= '`if`' ⟨*expression*⟩ '`then`' ⟨*expression*⟩ '`else`' ⟨*expression*⟩

⟨*forexpr*⟩ ::= '`for`' ⟨*type*⟩ ⟨*identifier*⟩ '`=`' ⟨*expression*⟩ '`,`'
⟨*expression*⟩ ['`,`' ⟨*expression*⟩] '`do`' ⟨*expression*⟩

⟨*withexpr*⟩ ::= '`with`' ⟨*type*⟩ ⟨*identifier*⟩ ['`=`' ⟨*expression*⟩]
('`,`' ⟨*type*⟩ ⟨*identifier*⟩ ['`=`' ⟨*expression*⟩])* '`do`' ⟨*expression*⟩

⟨*primary*⟩ ::= ⟨*identifierexpr*⟩
| ⟨*constexpr*⟩
| ⟨*parenexpr*⟩
| ⟨*ifexpr*⟩
| ⟨*forexpr*⟩
| ⟨*withexpr*⟩

⟨*unary*⟩ ::= ⟨*primary*⟩ | ⟨*unop*⟩ ⟨*unary*⟩

⟨*binoprhs*⟩ ::= (⟨*op*⟩ ⟨*unary*⟩)*

⟨*expression*⟩ ::= ⟨*unary*⟩ ⟨*binoprhs*⟩

⟨*prototype*⟩ ::= ⟨*type*⟩ ⟨*identifier*⟩ '`(`' [⟨*type*⟩ ⟨*identifier*⟩ ['`,`' ⟨*type*⟩ ⟨*identifier*⟩]*]
'`)`'

⟨*func_definition*⟩ ::= '`func`' ⟨*prototype*⟩ ⟨*expression*⟩

⟨*extern_definition*⟩ ::= '`extern`' ⟨*prototype*⟩

⟨*program*⟩ ::= (⟨*func_definition*⟩ | ⟨*extern_definition*⟩)*

Simbol ⟨*identifier*⟩ predstavlja ime spremenljivke ali funkcije v programu. To lahko vsebuje alfanumerične znake in podčrtaj, ne sme pa se začeti s številko (ali zapisano z regularnim izrazom: `[_A-Za-z][_A-Za-z0-9]*`).

Simbol ⟨*const*⟩ predstavlja konstantno vrednost, ki jo kot tako zna razpoznati kateri od modulov za tipe. Pregledovalnik pri ugotavljanju, ali gre za konstanto ali ne, kliče funkcijo za razpoznavanje konstantne vrednosti v vsakem modulu, ki jo ima.

Simbol ⟨*type*⟩ predstavlja ključno besedo (ang. *keyword*) kateregakoli tipa. Te besede so definirane po nalaganju vseh modulov za tipe, vsak modul ima svojo.

Komentarji so v jeziku predstavljeni z znakom `#` in segajo od znaka do konca vrstice.

Operatorji	Precedenca	
;	1	(veže najšibkeje)
=	2	
or	6	
xor	7	
and	8	
==, <>	9	
<, >, <=, >=	10	
+, -	20	
*, /	40	(veže najmočneje)

Tabela 3.2: Precedenca binarnih operatorjev.

Prednost binarnih operatorjev je prikazana v tabeli 3.2. Edina unarna operatorja, ki sta trenutno na voljo, sta - (unarni minus) in `not` (logična negacija).

3.4 Prevajalnik

Prevajalnik za jezik iz prejšnjega podpoglavja uporablja ročno narejen razpoznavalnik z rekurzivnim sestopom, za aritmetične izraze pa precedenčni razpoznavalnik. Z njima ustvari sintaksno drevo, iz katerega generira vmesno kodo za LLVM. Na njej LLVM izvede optimizacije in jo prevede v strojno kodo, ki se nahaja v pomnilniku. Preostanek sistema lahko dobi kazalec na katerokoli prevedeno funkcijo, ki jo potem izvede neposredno iz pomnilnika. Generirana strojna koda je hitrostno primerljiva z optimizirano C kodo.

Inicializacija prevajalnika. Prevajalnik je implementiran kot C++ razred. Narejen je tako, da lahko hkrati uporabljamo več instanc razredov — vsaka ima svoj kontekst, zato se med seboj ne motijo. Ob inicializaciji prevajalnik najprej inicializira LLVM generator strojne kode za procesor, na katerem teče. Potem nastavi tabelo s precedenco operatorjev, kot je prikazano v tabeli 3.2 na strani 17.

Zatem naloži module za tipe iz mape na disku in generira matriko operatorjev glede na operatorje, ki so implementirani v modulih. Matrika operatorjev preslika leksikalno besedo operatorja (po navadi je to kar znak, npr. +) in obeh tipov (v primeru binarnih operatorjev, unarni imajo samo enega) v kazalec na funkcijo iz ustreznega modula. Ta funkcija ob klicu generira LLVM vmesno kodo za ustrezni operator.

Potem inicializira knjižnico osnovnih funkcij. To vključuje predvsem interne funkcije (npr. za dodeljevanje pomnilnika) in tudi funkcije iz knjižnic posameznih modulov. Poleg tega naredi tudi konstruktorje za generatorje konstant v grafu operacij (več o tem v poglavju Izvajanje na strani 25).

Razpoznavalnik z rekurzivnim sestopom je po strukturi implementacije zelo podoben gramatiki, ki jo razpoznava. Sestoji iz vzajemno rekurzivnih funkcij, kjer vsaka funkcija implementira eno pravilo gramatike [35]. Razpoznavanje poteka od zgoraj navzdol (t.j. začne se pri najbolj splošnem pravilu, ki je po navadi kar pravilo za cel program). Za $LL(k)$ gramatike se lahko sestavi tak razpoznavalnik, ki teče v linearnem času.

Iz posameznega pravila se ustvarijo vozlišča sintaksnega drevesa. Posamezno vozlišče vsebuje podatke, ki so povezani z ustreznim pravilom. Na primer, vozlišče za binarno operacijo ima polje, ki pove, za katero operacijo gre (npr. seštevanje, množenje itd.), ter kazalca na levi in desni podizraz (t.j. izraza, ki sta levo in desno od operatorja).

S tem dobimo drevesno predstavitev izvorne kode.

Precedenčni razpoznavalnik od spodaj navzgor intepretira izraze, ki vsebujejo operatorje z različnimi prednostmi. Primeren je predvsem za razpoznavanje aritmetičnih izrazov.

Najboljši algoritem za implementacijo takega razpoznavalnika je t.i. *precedence climbing method* [17]. Aritmetične operacije lahko asociativno razpoznavamo od leve proti desni dokler imajo operatorji isto precedenco. Ko pridemo do operatorja z višjo precedenco, naredimo še en klic funkcije, le da tokrat sprejmemo samo operatorje z enako ali višjo precedenco.

Ta algoritem je namenjen uporabi skupaj s kakim močnejšim razpoznavalnikom (le-ta je potreben za razpoznavanje vseh ostalih elementov jezika, vključno z oklepaji za izraze).

Generiranje vmesne kode za LLVM. Vsak tip vozlišča v sintaksnem drevesu ima tudi metodo za generiranje LLVM vmesne kode. Ko je sintaksno drevo zgrajeno do konca, se na korenskem vozlišču pokliče metoda za generiranje kode, ki nadalje pokliče metode otrok.

Pri unarnih in binarnih operatorjih se kličejo funkcije iz modulov za tipe. Za primer implementacije take funkcije glej podpoglavje Jezik na strani 13.

Optimizacije. Po uspešnem zaključku generiranja vmesne kode se na njej izvede zaporedje optimizacijskih transformacij.

Zaporedje optimizacij je prikazano spodaj in temelji na zaporedju, ki ga izvede *clang* (to je prevajalnik za C, ki uporablja LLVM) pri najvišji stopnji optimizacije. Ker naš jezik nima kazalcev, si lahko privoščimo še dodatne optimizacije, ki jih *clang* ne izvede.

```
1     // Set up the optimizer pipeline
2     PassManager pm;
3
4     // Let passes know how the target lays out data structures
5     pm.add(new (GC) DataLayout(*ee->getDataLayout()));
6     // Simplify the control flow graph (deleting unreachable blocks, etc.)
7     pm.add(createCFGSimplificationPass());
8     // Promote allocas to registers
9     pm.add(createPromoteMemoryToRegisterPass());
10    // Eliminate obvious common subexpressions
11    pm.add(createEarlyCSEPass());
12    // Provide type-based alias analysis
13    pm.add(createTypeBasedAliasAnalysisPass());
14    // Do interprocedural constant propagation
15    pm.add(createIPConstantPropagationPass());
16    // Clean up after previous step
17    pm.add(createInstructionCombiningPass());
18    pm.add(createCFGSimplificationPass());
19    // Deduce function attributes
```

```
20      pm.add(createFunctionAttrsPass());
21      // Inline small functions
22      pm.add(createFunctionInliningPass());
23      // Promote args passed by-reference to by-value if they’re small enough
24      pm.add(createArgumentPromotionPass());
25      // Thread jumps
26      pm.add(createJumpThreadingPass());
27      // Propagate conditionals
28      pm.add(createCorrelatedValuePropagationPass());
29      // Simplify the control flow graph again
30      pm.add(createCFGSimplificationPass());
31      // Do simple "peephole" and bit-twiddling opts
32      pm.add(createInstructionCombiningPass());
33      // Eliminate tail calls
34      pm.add(createTailCallEliminationPass());
35      // Simplify the control flow graph again
36      pm.add(createCFGSimplificationPass());
37      // Reassociate expressions
38      pm.add(createReassociatePass());
39      // Convert loops into loop-closed SSA form
40      pm.add(createLCSSAPass());
41      // Simplify loop induction variables
42      pm.add(createIndVarSimplifyPass());
43      // Recognize loop idioms (e.g. memset)
44      pm.add(createLoopIdiomPass());
45      // Rotate loops
46      pm.add(createLoopRotatePass());
47      // Do loop invariant code motion (hoist loop invariants)
48      pm.add(createLICMPass());
49      // Unswitch loops (but not if it increases code size too much)
50      pm.add(createLoopUnswitchPass(true));
51      // Do simple "peephole" and bit-twiddling opts
52      pm.add(createInstructionCombiningPass());
53      // Do strength reduction on loop variables
54      pm.add(createLoopStrengthReducePass());
55      // Remove dead loops
56      pm.add(createLoopDeletionPass());
57      // Unroll small loops
58      pm.add(createLoopUnrollPass());
59      // Clean up after unrolling
60      pm.add(createInstructionCombiningPass());
61      // Eliminate common subexpressions and other redundancies
62      pm.add(createGVNPass());
63      // Do interprocedural sparse conditional constant propagation
64      pm.add(createIPSCCPPass());
65      // Redo a few things after redundancy elimination
66      pm.add(createInstructionCombiningPass());
```

```
67      pm.add(createJumpThreadingPass());
68      pm.add(createPromoteMemoryToRegisterPass());
69      pm.add(createCorrelatedValuePropagationPass());
70      pm.add(createAggressiveDCEPass());
71      // Vectorize basic blocks
72      pm.add(createBBVectorizePass());
73      pm.add(createInstructionCombiningPass());
74      pm.add(createGVNPass());
75      // Remove dead code
76      pm.add(createAggressiveDCEPass());
77      // Clean up
78      pm.add(createCFGSimplificationPass());
79      pm.add(createInstructionCombiningPass());
80
81      // Run all passes on generated code
82      pm.run(*mod);
```

Tukaj se vidi prednost uporabe sistema LLVM. LLVM je v razvoju že od leta 2000 in nudi veliko kompleksnih optimizacij. Samostojna implementacija vseh zgornjih optimizacij bi presegala čas, ki je na voljo za izdelavo diplomske naloge.

3.5 Primeri kode

Vsi konstrukti v razvitem jeziku so funkcijski izrazi, zato jih lahko kombiniramo na enak način kot aritmetične izraze. Jezik ni občutljiv na število presledkov, zamikanje in nove vrstice.

Za prvi primer si poglejmo rekurzivno implementacijo računanja n-tega Fibonaccijevega števila v razvitem jeziku:

```
func int fib(int n) if n < 3 then 1 else fib(n - 1) + fib(n - 2)
```

V jeziku C bi zgornja funkcija izgledala takole:

```
int fib(int n) { return n < 3 ? 1 : fib(n - 1) + fib(n - 2); }
```

Kot vidimo, sta si jezika na prvi pogled dokaj podobna. Glavna razlika je ta, da je v tem jeziku vse funkcijski izraz, tako da `return` ni potreben. Sintaksa je preprostejša in skupaj z uporabo angleških besed poenostavi tako razumevanje, kot tudi pisanje programov.

Malce bolj berljiv zapis C variante:

```
int fib(int n)
{
    if (n < 3)
        return 1;

    return fib(n - 1) + fib(n - 2);
}
```

V jeziku Scheme je ekvivalentna funkcija taka:

```
(define fib
    (lambda (n)
        (if (< n 3)
        1
        (+ (fib (- n 1)) (fib (- n 2))))))
```

Iterativna verzija z realnimi števili v razvitem jeziku izgleda tako:

```
func real fibr(real n)
    with real a = 1.0, real b = 1.0, real c do  # deklaracija spremenljivk
        (for real i = 3.0, i < n do
            c = a + b;
            a = b;
            b = c);
        b
```

Z `with` deklariramo spremenljivke, ki so veljavne v izrazu, ki sledi besedi `do`. Zanka `for` je obdana v oklepaje zaradi prioritete operatorja `;`. Ta operator se obnaša tako kot `,` v jeziku C — evaluira levo in desno stran ter kot rezultat vrne rezultat desne strani.

V jeziku C bi ta varianta izgledala takole:

```
1  double fibr(double n)
2  {
3      double a = 1.0, b = 1.0, c, i;
4
5      for (i = 3.0; i < n; i += 1.0) {
6          c = a + b;
7          a = b;
8          b = c;
9      }
10
11     return b;
12 }
```

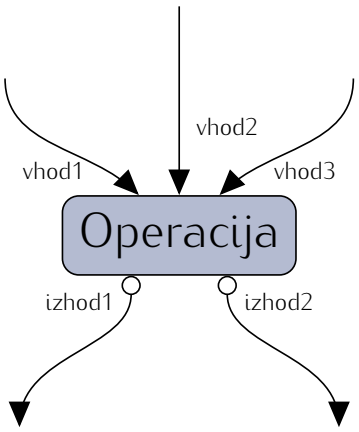
POGLAVJE 4

Izvajanje

Namen te komponente je izvajanje podatkovno-pretočnih grafov (v nadaljevanju: PPG).

Uvod v grafe. PPG vsebujejo vozlišča, ki predstavljajo operacije, in povezave med njimi, po katerih se pretakajo podatki. V našem primeru so PPG usmerjeni in aciklični, ker to poenostavi povzporejanje izvajanja in ker za veliko večino namenov uporabe sistema cikli sploh niso potrebni.

Vsako vozlišče ima lahko vhodna in izhodna vrata (slika 4.1). Na vrata se vežejo povezave do drugih vozlišč. Pri posameznem vozlišču morajo biti vsa vhodna vrata povezana na izhodna vrata nekih drugih vozlišč, izhodna vrata tega vozlišča pa lahko ostanejo nepovezana.



Slika 4.1: Prikaz vozlišča (operacije) v podatkovno-pretočnem grafu.

Vhodna vrata posameznega vozlišča so zgolj kazalci na izhodna vrata drugih vozlišč. Izhodna vrata pa vsebujejo tudi podatke, prostor zanje si pridobi vozlišče ob instanciranju.

Vozlišča brez vhodov so po navadi generatorji konstant, vozlišča brez izhodov pa po navadi shranijo oz. prikažejo končni rezulat.

Vsaka vrata imajo določen tip za podatke, ki jih sprejemajo oz. vsebujejo. Če se tipa vhodnih in izhodnih vrat ne ujemata, ne moremo ustvariti povezave med njima.

Uvod v izvajanje. Ob izvajanju PPG dobi vsaka operacija parameter, ki ji pove, za kateri trenutek ($t \in \mathbb{N}_0$) naj izvede tisto operacijo. To pride prav pri izvajanju grafov, ki generirajo animacije (v tem primeru je t enak zaporedni številki sličice animacije).

Posamezen PPG lahko izvedemo na več načinov. Trenutno so implementirani sledeči:

- **enkratno izvajanje** — izvedi graf natanko enkrat za določen t,
- **večkratno izvajanje** — izvedi graf za več različnih t-jev (npr. pri generiranju animacij),
- **neprekinjeno izvajanje** — izvajaj graf do prekinitve s strani uporabnika (t se začne pri 0 in monotono narašča).

Pri izvajanju grafa je uporabljen lasten razporejevalnik, opisan v podpoglavju Razporejevalnik na strani 28. Generiranje opravil za razporejevalnik iz grafa je opisano v podpoglavju Izvajalnik na strani 29.

Operacije. Operacije so lahko implementirane v kateremkoli programskem jeziku, ki podpira prevajanje kode v dinamično naložljive module (ang. *shared object*, *.so*). Lahko pa so implementirane tudi v jeziku, opisanem v poglavju Jezik na strani 9.

Moduli za operacije morajo implementirati vsaj dve funkciji: `instance_init(n)` in `instance_action(n, t)`. Prva ustvari vhode in izhode vozlišča (ob tem definira, katere tipe pričakuje). Druga pa izvede operacijo za podan trenutek `t`. Poleg tega lahko modul implementira še funkcijo `instance_verify_inputs(n)`, ki se kliče pred izvajanjem operacije. V tej funkciji lahko preveri, ali vhodni podatki ustrezajo omejitvam.

Kot pri modulih za tipe, opisanih v podpoglavju Jezik na strani 13, so tudi tukaj bili razviti makri za lažje pisanje modulov. Implementacija le-teh se nahaja v datoteki `ops.h`.

Za primer si poglejmo modul za gama korekcijo slik:

```
1  #include "ops.h"
2  #include "../types/image.h"
3
4  #include <math.h>
5
6  DECLARE_NODE("CorrectGamma")
7
8  INSTANCE_INIT()
9  {
10     node_create_input(node,  "imageIn",  "image");
11     node_create_input(node,  "gamma",    "real");
12
13     node_create_output(node, "imageOut", "image");
14
```

```
15      return 1;
16  }
17
18  INSTANCE_ACTION()
19  {
20      const Image *const in = (const Image *) node_get_input_data(node, "imageIn");
21      const double       g  = *(const double *) node_get_input_data(node, "gamma");
22
23      Image *out = new (GC) Image(in->width, in->height);
24
25      const size_t npix = in->width * in->height;
26
27      const double G = 1.0 / g;
28
29      for (size_t i = 0; i < npix; i++) {
30          out->data[i].r = pow(in->data[i].r, G);
31          out->data[i].g = pow(in->data[i].g, G);
32          out->data[i].b = pow(in->data[i].b, G);
33          out->data[i].a = in->data[i].a;
34      }
35
36      node_set_output_data(node, "imageOut", (void *)out);
37
38      return 1;
39  }
```

Na začetku se z `DECLARE_NODE` definira ime operacije, v `INSTANCE_INIT()` se določi katere vhode in izhode ima operacija ter kakšne tipe pričakuje, v `INSTANCE_ACTION()` pa je dejanska implementacija operacije. Kaj delajo pomožne funkcije, bi moralo biti razvidno iz njihovih imen.

V namenskem jeziku lahko nove operacije pišemo na podoben način. V mapo z moduli za operacije lahko poleg `.so` datotek damo tudi datoteke s končnico `.op`. Te datoteke se lahko nahajajo tudi v mapi s trenutno izvajajočo skripto, naložile se bodo pred prevajanjem skripte.

V posamezni *imemodula*`.op` datoteki je lahko definirana največ ena operacija. S funkcijama *imemodula*`_instance_init(n)` in *imemodula*`_instance_action(n, t)` definiramo operacijo. Pomožne funkcije delujejo na enak način, le da imajo malce drugačna imena — ker je jezik strogo tipiran, se za vsak podatkovni tip generirata funkciji `node_get_input_data_`*imetipa*`()` in `node_set_output_data_`*imetipa*`()`.

4.1 Razporejevalnik

Glavni namen razporejevalnika je razporejanje opravil med procesorska jedra. Glede na to, kako to dela, poznamo več vrst razporejevalnikov. V grobem se delijo na prekinljive in neprekinljive [19]. Pri prekinljivih dobi vsako opravilo določen del procesorskega časa, v katerem se lahko izvaja, potem pa se izvajanje opravila prekine in se na njegovem mestu začne izvajati drugo opravilo. Po določenem času se ta postopek ponovi. Tak pristop je primeren za razporejevalnike v splošnonamenskih operacijskih sistemih, saj daje vtis, da hkrati teče več procesov. Neprekinljivi razporejevalniki dodelijo opravilo nekemu jedru, ki ga potem izvaja, dokler se opravilo ne konča.

Razporejevalnik, razvit v diplomski nalogi, je zadolžen za neprekinljivo izvajanje poljubnih opravil na več jedrih oz. procesorjih. Vsako opravilo ima določeno prioriteto, predstavljeno s 64-bitno celoštevilsko vrednostjo. Manjša številka predstavlja višjo prioriteto (torej, opravilo s prioriteto 3 se bo izvedlo pred vsemi opravili prioritete 4).

Ob inicializaciji programa se naredi toliko niti, kot je vseh procesorskih jeder v sistemu. Vsako nit se pripne na svoje jedro (to je namig razporejevalniku operacijskega sistema, da naj jih ne premika na druga jedra). Niti so ustvarjene s standardno POSIX implementacijo niti (*pthreads*).

Razporejevalnik ima eno prioritetno vrsto, ki vsebuje morebitna opravila za obdelavo, in množico opravil, ki so trenutno v izvajanju.

Niti razporejevalnika pregledujejo prioritetno vrsto in iz nje jemljejo opravila za obdelavo. Če je v množici izvajajočih se opravil kakšno opravilo z višjo prioriteto od trenutnega opravila iz prioritetne vrste, ga nit ne bo vzela v izvajanje.

V prioritetno vrsto se lahko kadarkoli dodajajo nova opravila. Tudi med izvajanjem kakega že obstoječega opravila.

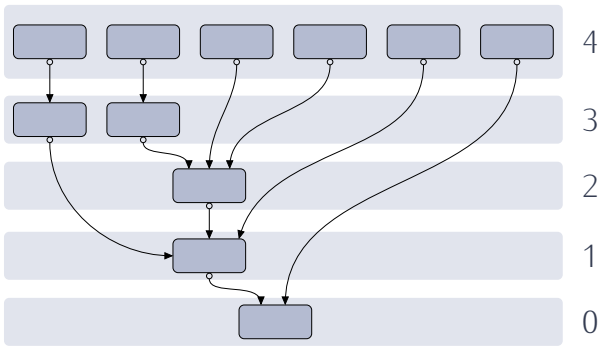
Razporejevalnik nudi tudi funkcijo za čakanje na zaključek opravil z določeno prioriteto oz. višjo.

Funkcionalnost razporejevalnika bi bilo mogoče implementirati s kakšno že obstoječo tehnologijo, na primer OpenMP. V poglavju Zaključek na strani 53 so opisane načrtovane izboljšave, iz katerih bi moralo biti razvidno, zakaj to ne bi bilo smiselno.

4.2 Izvajalnik

Izvajalnik pretvori graf v opravila za razporejevalnik. Vsakemu opravilu dodeli prioriteto, glede na odvisnosti iz grafa.

Graf lahko razbijemo na ravni (slika 4.2). Operacije na posamezni ravni so med seboj neodvisne in jih zato lahko izvedemo hkrati.



Slika 4.2: Prikaz razbitja grafa na ravni.

Koraki pri ustvarjanju opravil iz grafa so sledeči:

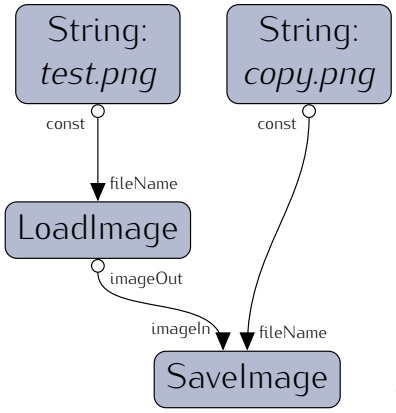
1. Preveri, ali so vhodi vseh vozlišč v grafu povezani na izhode vozlišč z ustreznim tipom — če niso, se sledeči koraki ne izvedejo.
2. Ustvari polje množic vozlišč. To predstavlja ravni grafa z operacijami. Na začetku imamo samo raven 0, ki je prazna.
3. Najdi končna vozlišča (to so vozlišča brez izhodov) in jih postavi na raven 0.
4. Sedaj je potrebno zapolniti višje ravni s predniki vozlišč z ravni 0:
   (a) V zanki ponavljaj:
      i. Ustvari novo raven.
      ii. Zapolni ustvarjeno raven z vozlišči, na katera so povezana vozlišča s prejšnje ravni.
      iii. Če ni bilo dodanih nobenih novih vozlišč, zbriši raven in končaj zanko.
5. Sprehodimo se po ravneh (od najvišje k 0) in operacije kot opravila dodamo razporejevalniku. Prioriteta opravil z iste ravni grafa je $N - i$, kjer je N število vseh ravni, i pa številka trenutne.

4.3 Ustvarjanje grafov

Grafe lahko ustvarjamo v jeziku, opisanem v poglavju Jezik na strani 9. Graf je v sistemu predstavljen globalno.

Za vsako naloženo operacijo se naredi konstruktor, s katerim dobimo vrednost tipa `node`. Ob klicu konstruktorja se novo vozlišče avtomatsko doda v graf. Posamezne operacije povezujemo s funkcijo `connect(vozlisce1, "imeIzhoda", vozlisce2, "imeVhoda")`. Generatorji konstant imajo konstruktor, ki sprejme še dodaten argument, in sicer vrednost konstante.

Poglejmo si preprost primer skripte, ki ustvari graf, ki zgolj skopira sliko:




```
1  func int main()
2      (with node li, node si, node fi, node fo do
3          fi = String("test.png");
4          fo = String("copy.png");
5          li = LoadImage();
6          si = SaveImage();
7          connect(fi, "const",    li, "fileName");
8          connect(li, "imageOut", si, "imageIn");
9          connect(fo, "const",    si, "fileName"));
10     0
```

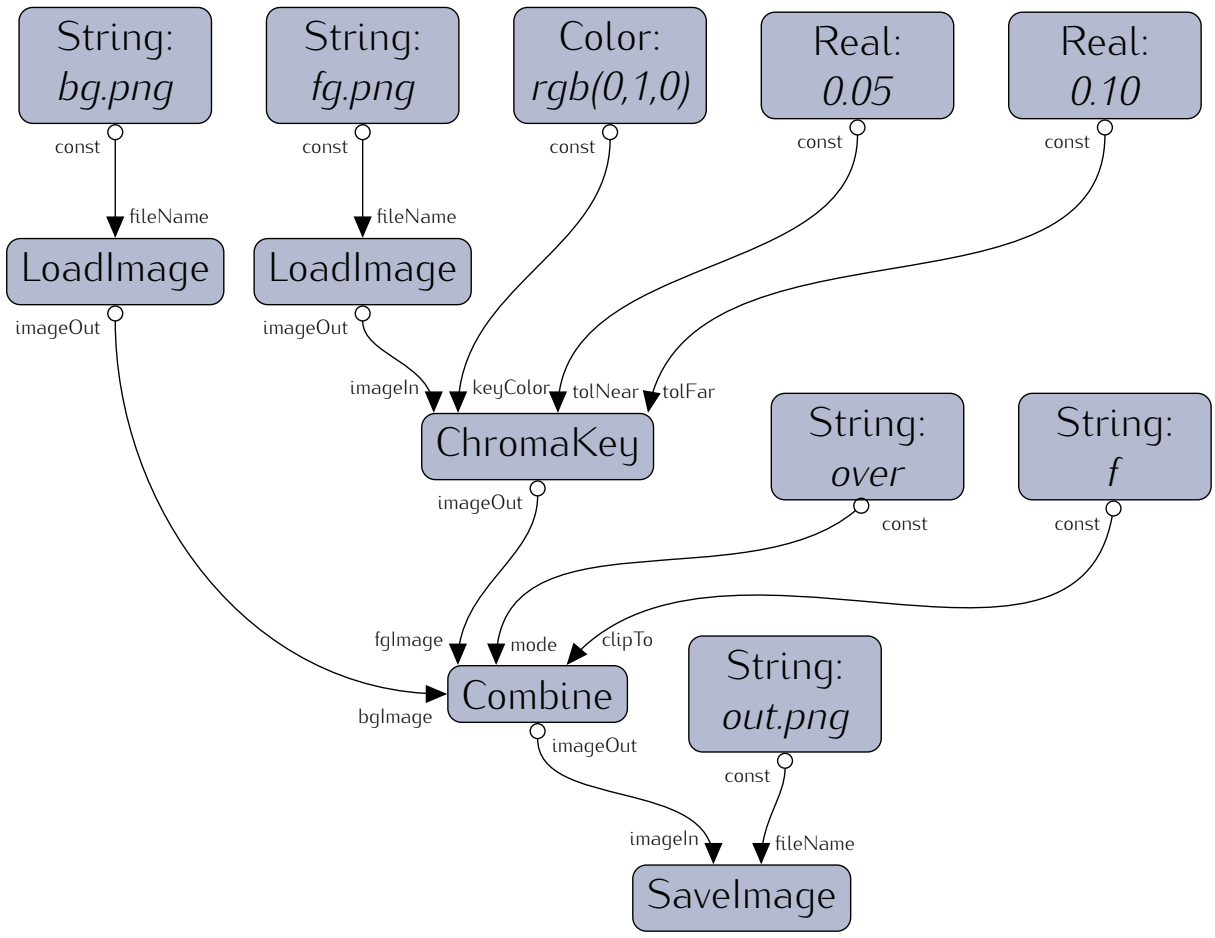
Slika 4.3: Primer grafa (kopiranje slike).

Zgornja koda ustvari graf na sliki 4.3.

Funkcija `connect()` je zadolžena za preverjanje, ali so v grafu cikli ali ne. Če pride do cikla, se izvajanje skripte ustavi, napaka pa se javi uporabniku.

Poglejmo si še malce bolj zahteven primer. Koda generira graf s slike 4.4. Najprej ustvari vozlišča, ki so avtomatsko dodana v graf, ter njihova vrata poveže med seboj.

Graf naloži dve sliki, `bg.png` in `fg.png`. Na `fg.png` naredi zeleno barvo prosojno, rezultat pa položi čez sliko `bg.png` in končni kompozit shrani v datotetko `out.png`.



Slika 4.4: Primer grafa (kompozitiranje slik).

```
1  func int main()
2      with node s1, node s2, node s3, node s4,
3           node s5, node c1, node r1, node r2,
4           node li1, node li2, node ck1,
5           node cb1, node si1
6      do (
7          # Ustvarimo generatorje konstant
8          s1 = String("bg.png");
9          s2 = String("fg.png");
10         c1 = Color(color(0.0, 1.0, 0.0, 1.0));
11         r1 = Real(0.05);
12         r2 = Real(0.10);
13         s3 = String("over");
14         s4 = String("f");
15         s5 = String("out.png");
16
17         # Ustvarimo operacije
18         li1 = LoadImage();
19         li2 = LoadImage();
20         ck1 = ChromaKey();
21         cb1 = Combine();
22         si1 = SaveImage();
23
24         # Povezemo vrata vozlisc med seboj
25         connect(s1,  "const",    li1, "fileName");
26         connect(s2,  "const",    li2, "fileName");
27         connect(c1,  "const",    ck1, "keyColor");
28         connect(r1,  "const",    ck1, "tolNear");
29         connect(r2,  "const",    ck1, "tolFar");
30         connect(li2, "imageOut", ck1, "imageIn");
31         connect(s3,  "const",    cb1, "mode");
32         connect(s4,  "const",    cb1, "clipTo");
33         connect(ck1, "imageOut", cb1, "fgImage");
34         connect(li1, "imageOut", cb1, "bgImage");
35         connect(cb1, "imageOut", si1, "imageIn");
36         connect(s5,  "const",    si1, "fileName");
37     );
38     0
```

POGLAVJE 5

Področja uporabe

V tem poglavju je opisanih nekaj področij uporabe sistema ter operacije, ki so bile implementirane za posamezno področje.

Pri opisu posamezne operacije je še tabela, v kateri so opisana vrata vozlišča. Vsaka vrata imajo ime, podatkovni tip in opis. Imena vhodnih vrat so v modri barvi, izhodnih pa v rdeči. V stolpcu V/I je črka V, če gre za vhodno vozlišče, ali I za izhodno. Primer je prikazan v tabeli 5.1.

ImeOperacije

V/I	Ime vrat	Tip	Opis
V	`vhodnaVrata1`	`tip1`	Opis vrat 1
V	`vhodnaVrata2`	`tip2`	Opis vrat 2
I	`izhodnaVrata`	`tip`	Opis vrat

Tabela 5.1: Primer tabele z opisom vrat vozlišča operacije.

Vozlišča imajo lahko več vhodnih in več izhodnih vrat. Podatkovni tipi, ki so na voljo, so predstavljeni v tabeli 3.1 (v podpoglavju Jezik na strani 15).

5.1 Kompozitiranje

5.1.1 CorrectGamma

CorrectGamma

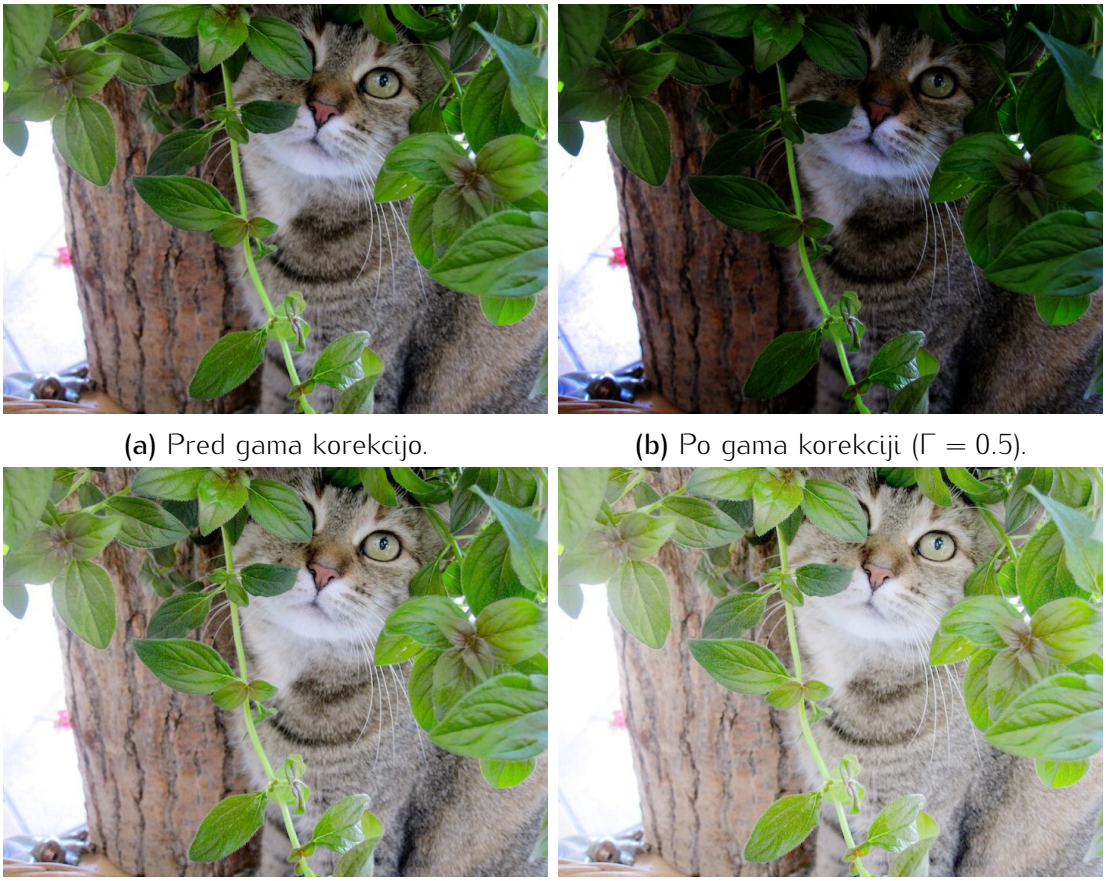
V/I	Ime vrat	Tip	Opis
V	`imageIn`	`image`	Vhodna slika
V	`gamma`	`real`	Gama faktor
I	`imageOut`	`image`	Izhodna slika

Operacija opravi gama korekcijo na vhodni sliki. Za vsako točko P v vhodni sliki izvede operacijo [4]:

$$P_{rgb} = P_{rgb}^{1/\Gamma} \qquad [5.1]$$

Alfa kanal ostane nespremenjen.

Na sliki 5.1 je prikazan primer uporabe te operacije.



(a) Pred gama korekcijo. **(b)** Po gama korekciji ($\Gamma = 0.5$). **(c)** Po gama korekciji ($\Gamma = 1.5$). **(d)** Po gama korekciji ($\Gamma = 2.2$).

Slika 5.1: Primer gama korekcije.

5.1.2 Combine

Operacija `Combine` implementira najpomembnejšo operacijo kompozitiranja — kombiniranje slik.

Pred kombiniranjem slik morajo vhodne slike imeti barvne komponente pomnožene z vrednostjo alfa kanala (t.i. *premultiplied alpha*). Tak pristop uporabljajo tudi druga orodja za kompozitiranje, saj poenostavi operacijo kombiniranja, poleg tega pa postanejo te operacije asociativne [15, 4].

Combine

V/I	Ime vrat	Tip	Opis
V	`fgImage`	`image`	Slika ospredja
V	`bgImage`	`image`	Slika ozadja
V	`mode`	`string`	Način kombiniranja slik
V	`clipTo`	`string`	Velikost izhodne slike enaka ospredju (`f`) ali ozadju (`b`).
I	`imageOut`	`image`	Izhodna slika

Možni načini (`mode`) kombiniranja slik so prikazani v tabeli 5.2, v vsaki vrstici je podan tudi primer. Originalni sliki sta  in  (vir: [9]), primeri so narejeni z implementirano operacijo.

Način	Rezultat	Primer
`add`	$A + B$	
`sub`	$A - B$	
`mul`	$A \times B$	
`div`	$A \div B$	
`in`	$A \times B_\alpha$	
`out`	$A \times (1 - B_\alpha)$	
`over`	$A + B \times (1 - A_\alpha)$	
`atop`	$A \times B_\alpha + B \times (1 - A_\alpha)$	
`min`	$\min(A, B)$	
`max`	$\max(A, B)$	
`xor`	$A \times (1 - B_\alpha) + B \times (1 - A_\alpha)$	
`screen`	$1 - (1 - A) \times (1 - B)$	

Tabela 5.2: Podprti načini kombiniranja slik.

5.1.3 ToneMap

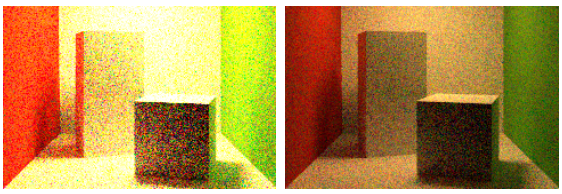
ToneMap

V/I	Ime vrat	Tip	Opis
V	`imageIn`	`image`	Vhodna slika
I	`imageOut`	`image`	Izhodna slika

`ToneMap` izvede mapiranje odtenkov na vhodni sliki.

Na današnjih napravah za prikaz slik ni mogoče natančno predstaviti zelo svetlih ali zelo temnih področij slik, ki so računalniško generirane ali zajete s senzorji. Take slike imajo mnogo večji dinamični razpon, kot pa ga je zaslon zmožen prikazati.

Mapiranje odtenkov preslika barvne komponente posamezne točke na sliki na interval $[0, 1]$, ki ga lahko prikažemo na zaslonu, in pri tem ohrani dinamični razpon slike. Za to je uporabljen Wardov algoritem [20].



(a) Brez mapiranja. **(b)** Z mapiranjem.

Slika 5.2: Primer mapiranja odtenkov.

Slednji izvede sledeče korake:

1. najprej izračuna logaritemsko povprečje svetilnosti točk P vhodne slike I, ki ima N točk:

$$L_{logavg} = e^{\frac{1}{N}\sum_{\forall P \in I} \log\left(0.2126P_r + 0.7152P_g + 0.0722P_b\right)} \quad [5.2]$$

   ($0.2126P_r + 0.7152P_g + 0.0722P_b$ je skalarni produkt vektorja barve z vektorjem koeficientov za pretvorbo RGB barve v fotometrično svetilnost [30]; zelena ima največji koeficient, ker so človeške oči na to barvo najbolj občutljive);

2. potem izračuna skalirni faktor za vse točke:

$$f = \frac{1}{L_{dmax}} \left(\frac{1.219 + \left(\frac{1}{2} L_{dmax}\right)^{0.4}}{1.219 + L_{logavg}^{0.4}} \right)^{2.5} \quad [5.3]$$

   (L_{dmax} je največja svetilnost zaslona; po sRGB standardu je ta konstanta enaka $80\frac{cd}{m^2}$, v praksi zasloni dosegajo svetilnosti do $300\frac{cd}{m^2}$ in celo več [21]);

3. na koncu vsako točko slike pomnoži s skalirnim faktorjem f:

$$P_{rgb} = P_{rgb} \cdot f, \quad [5.4]$$

   alfa kanal ostane nespremenjen.

Učinek mapiranja odtenkov je prikazan na sliki 5.2.

5.1.4 ChromaKey

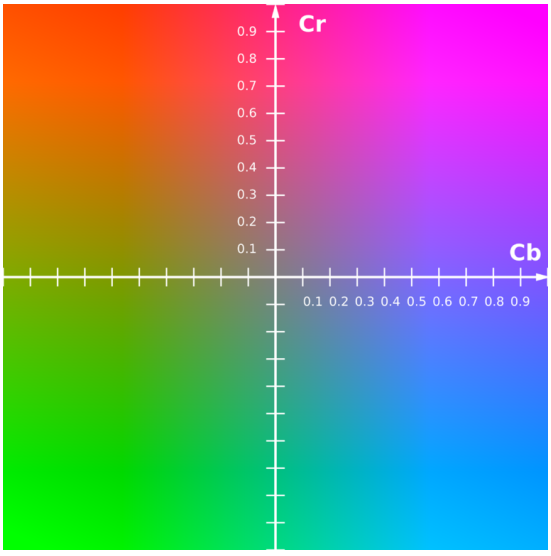
Ta operacija naredi vse točke določene barve (znotraj podane tolerance) prosojne.

ChromaKey

V/I	Ime vrat	Tip	Opis
V	`imageIn`	`image`	Vhodna slika
V	`keyColor`	`color`	Barva ozadja
V	`tolNear`	`real`	Bližnja toleranca
V	`tolFar`	`real`	Daljna toleranca
I	`imageOut`	`image`	Izhodna slika

Uporabljen je dokaj preprost algoritem [5]. Najprej pretvorimo vhodno sliko in barvo ozadja v barvni prostor YCbCr (ločimo svetilnost Y od barvnih komponent Cb in Cr) in delamo zgolj s Cb in Cr. Komponenti predstavljata ravnino (slika 5.3), na kateri lahko računamo razdalje posamezne točke slike od točke, ki predstavlja barvo ozadja.

Pri računanju razdalj uporabimo dve meji — bližnjo in daljno toleranco. S tem razdelimo ravnino na tri območja. Bližnje območje vsebuje barve, ki so zelo blizu barvi ozadja (t.j. so oddaljene za manj kot `tolNear`), daljno območje vsebuje barve, ki so od barve ozadja oddaljene več kot `tolFar`, med njima pa je vmesno območje. Točke slike z barvami, ki so v bližnjem območju, naredimo popolnoma prosojne (vrednost alfa kanala take točke je 0). Barve v daljnem območju so popolnoma vidne (vrednost alfa kanala je 1). V vmesnem območju pa moramo prosojnost interpolirati glede na oddaljenost od barve ozadja. Za boljše rezultate na koncu od dobljenega ospredja odštejemo barvo ozadja, uteženo z razdaljo od le-te. Primer je prikazan na sliki 5.4.



Slika 5.3: CbCr barvna ravnina pri $Y = 0.5$ (vir: [22]).

(a) Originalna slika.

(b) Slika z odstranjenim ozadjem (barva (0.490, 0.565, 0.243), toleranci 0.1 in 0.125).

Slika 5.4: Primer operacije `ChromaKey`.

5.1.5 DiffKey

DiffKey

V/I	Ime vrat	Tip	Opis
V	`imageIn`	`image`	Vhodna slika
V	`cleanPlate`	`image`	Slika ozadja
V	`tolNear`	`real`	Bližnja toleranca
V	`tolFar`	`real`	Daljna toleranca
I	`imageOut`	`image`	Izhodna slika

S to operacijo naredimo vse točke, ki se od slike ozadja razlikujejo za več kot podano toleranco, prosojne.

Implementacija je podobna kot pri `ChromaKey`, le da imamo tukaj sliko ozadja in ne zgolj ene barve.

5.1.6 Resize

Resize

V/I	Ime vrat	Tip	Opis
V	`imageIn`	`image`	Vhodna slika
V	`width`	`int`	Nova širina
V	`height`	`int`	Nova višina
I	`imageOut`	`image`	Izhodna slika

`Resize` poveča ali pomanjša vhodno sliko na dane dimenzije.

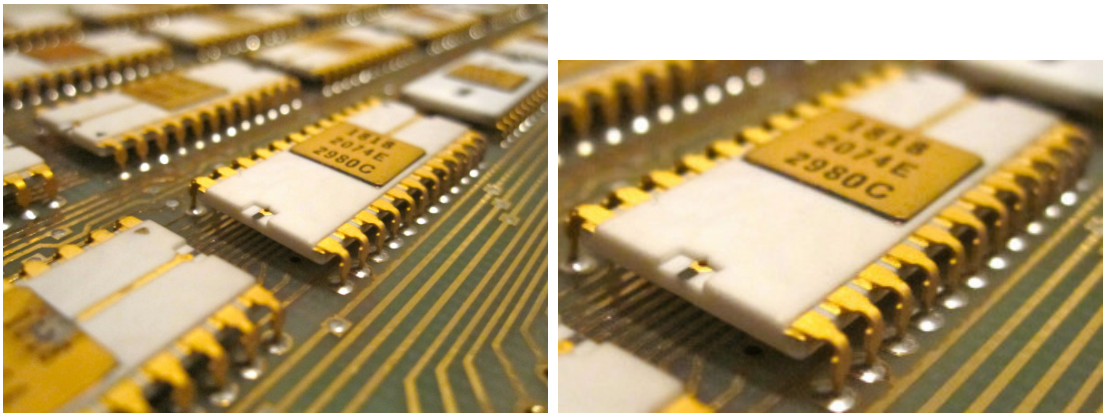
5.1.7 Crop

Crop

V/I	Ime vrat	Tip	Opis
V	`imageIn`	`image`	Vhodna slika
V	`left`	`int`	Začetni x položaj
V	`top`	`int`	Začetni y položaj
V	`right`	`int`	Končni x položaj
V	`bottom`	`int`	Končni y položaj
I	`imageOut`	`image`	Izhodna slika

`Crop` obreže sliko na podan okvir. Okvir je lahko večji od slike, v tem primeru bodo področja izven slike prosojna.

Primer je prikazan na sliki 5.5.



(a) Originalna slika (600 × 450 točk). **(b)** Izrez pravokotnika (200, 100, 540, 320).

Slika 5.5: Primer rezanja slike.

5.1.8 AffineTransform

Operacija `AffineTransform` opravi afino transformacijo na sliki.

Afine transformacije ohranijo ravne črte in razmerja razdalj med točkami, ki ležijo na njih. Vzporedne črte so še vedno vzporedne po afini transformaciji. Najpogostejše afine transformacije so premik, zrcaljenje in rotacija.

AffineTransform

V/I	Ime vrat	Tip	Opis
V	`imageIn`	`image`	Vhodna slika
V	`transform`	`mat3`	Transformacijska matrika
I	`imageOut`	`image`	Izhodna slika

Transformacijska matrika ima obliko:

$$\begin{bmatrix} a & b & x_0 \\ c & d & y_0 \\ 0 & 0 & 1 \end{bmatrix} \quad [5.5]$$

Parametra x_0 in y_0 predstavljata premik v x in y smeri, ostali parametri pa določajo transformacijsko matriko za razteg in rotacijo na sledeč način:

$$a = \boxed{\text{faktor raztega x}} \cdot \cos\left(\boxed{\text{rotacija x v stopinjah}} \cdot \frac{\pi}{180}\right) \quad [5.6]$$

$$b = \boxed{\text{faktor raztega y}} \cdot \sin\left(\boxed{\text{rotacija y v stopinjah}} \cdot \frac{\pi}{180}\right) \quad [5.7]$$

$$c = \boxed{\text{faktor raztega x}} \cdot \sin\left(\boxed{\text{rotacija x v stopinjah}} \cdot \frac{\pi}{180}\right) \quad [5.8]$$

$$d = \boxed{\text{faktor raztega y}} \cdot \cos\left(\boxed{\text{rotacija y v stopinjah}} \cdot \frac{\pi}{180}\right) \quad [5.9]$$

Pri transformiranju slike z matriko 5.5 se najprej izračuna njen inverz, potem pa se za vsako točko izhodne slike izračuna, kateri točki v originalni sliki ustreza [12]. Prednost tega pristopa je, da v izhodni sliki nimamo nobenih lukenj oz. prepisanih točk. Za boljši izgled končne slike uporablja ta operacija še bilinearno interpolacijo — vrednost vsake izhodne točke je uteženo povprečje štirih sosedov izračunane točke v vhodni sliki.

Primer uporabe te operacije je prikazan na sliki 5.6.

(a) Originalna slika.

(b) Po transformaciji z matriko:

$$\begin{bmatrix} \frac{1}{2}\cos\left(45\frac{\pi}{180}\right) & \frac{1}{2}\sin\left(-45\frac{\pi}{180}\right) & 0 \\ \frac{1}{2}\sin\left(45\frac{\pi}{180}\right) & \frac{1}{2}\cos\left(-45\frac{\pi}{180}\right) & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Slika 5.6: Primer afine transformacije.

5.1.9 Convolve2D

Convolve2D

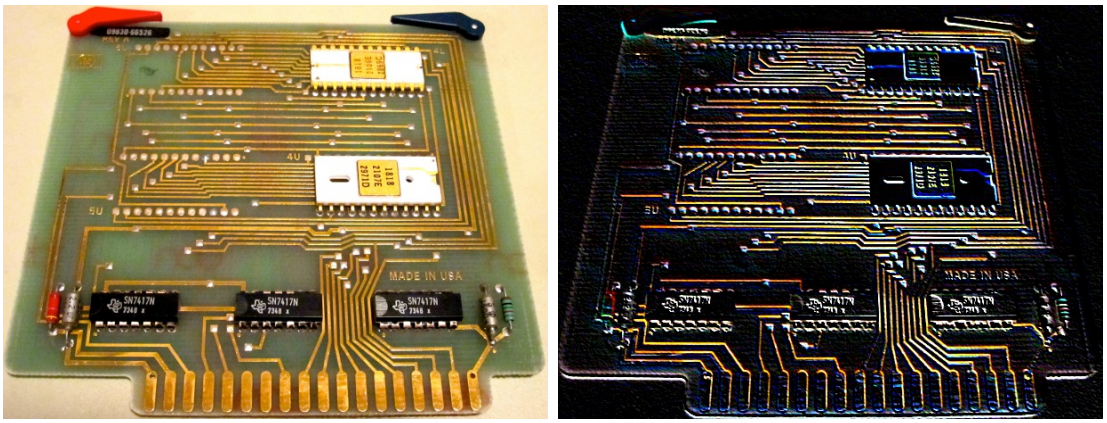
V/I	Ime vrat	Tip	Opis
V	`imageIn`	`image`	Vhodna slika
V	`kernel`	`mat3`	Konvolucijsko jedro
I	`imageOut`	`image`	Izhodna slika

`Convolve2D` izvede konvolucijo podanega jedra z vhodno sliko.

S konvolucijo slike z ustreznim jedrom (filtrom) lahko sliko izostrimo, zameglimo, ojačamo robove itd.

Konvolucija drsi z jedrom čez sliko in računa uteženo povprečje trenutne točke in njenih sosedov. Vrednost izhodne točke je to povprečje. Manjša težava nastane na robovih, saj le-ti nimajo sosednjih točk. Možna rešitev je, da izhodna slika ne vsebuje robov, torej je v obeh dimenzijah za 2 točki manjša. To zna biti problematično pri zaporedju konvolucij, zato pri tej operaciji robove enostavno skopiramo iz originalne slike. V primerih, kjer bi bili robovi moteči, lahko uporabnik še vedno uporabi operacijo `Crop` in jih odreže.

Trenutno je implementirana podpora za jedra velikosti 3×3, ki so najpogostejša.



(a) Originalna slika.

(b) Po konvoluciji z jedrom $\begin{bmatrix} 1 & 2 & 1 \\ -2 & 0 & 2 \\ -1 & -2 & -1 \end{bmatrix}$.

Slika 5.7: Primer konvolucije.

Na sliki 5.7 je prikazan primer konvolucije z jedrom, ki poudari robove.

5.2 Izrisovanje

5.2.1 Sphere

`Sphere` ustvari 3D sfero (t.j. plašč krogle) z danim radijem.

Sphere

V/I	Ime vrat	Tip	Opis
V	`radius`	`real`	Radij sfere
V	`center`	`vec3`	Položaj središča
V	`material`	`material`	Opis materiala
I	`object`	`obj`	Sfera

5.2.2 LoadTriangleMesh

Z `LoadTriangleMesh` naložimo 3D model iz datoteke. Modeli so trenutno lahko zgolj v OBJ formatu [33], vendar je sistem narejen tako, da se za podporo drugih formatov lahko doda nove vhodno-izhodne module, te operacije pa ni potrebno spreminjati.

LoadTriangleMesh

V/I	Ime vrat	Tip	Opis
V	`fileName`	`string`	Ime datoteke
V	`flipNormals`	`bool`	Obrni smer normal
V	`material`	`material`	Opis materiala
I	`object`	`obj`	Model

5.2.3 Camera

Operacija `Camera` na danem položaju ustvari kamero, ki gleda v neko točko v 3D prostoru. Poleg teh dveh parametrov moramo podati še smer, ki kaže navzgor (z vidika kamere). Slednjo rabimo zato, da enolično določimo rotacijo kamere (to bi lahko storili tudi z matriko, vendar je ta pristop bolj prijazen do uporabnika).

Camera

V/I	Ime vrat	Tip	Opis
V	`eye`	`vec3`	Položaj kamere
V	`lookAt`	`vec3`	Točka, kamor gleda kamera
V	`up`	`vec3`	Smer, ki kaže navzgor
I	`camera`	`camera`	Kamera

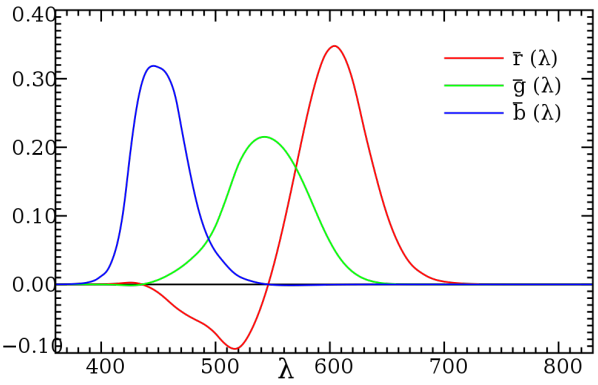
5.2.4 LoadSpectrum

LoadSpectrum

V/I	Ime vrat	Tip	Opis
V	`fileName`	`string`	Ime datoteke
I	`spectrum`	`spectrum`	Barvni spekter

`LoadSpectrum` iz datoteke naloži opis spektra valovnih dolžin.

Izrisovalnik 3D scen ne uporablja RGB barv, temveč spektre. Vsak opis materiala ima barvo definirano s spektrom valovnih dolžin in ne z deležem rdeče, zelene in modre barve. To omogoča bolj natančno predstavitev barvnih površin, saj v modelu RGB ne moremo predstaviti vseh barv, ki jih naše oči lahko zaznajo. Pri določenih spektrih barv, ki vsebujejo valovne dolžine med 400 in 550 nanometrov, bi rdeča komponenta morala biti negativna (glej sliko 5.8). Ker so vse barve predstavljene s spektri, lahko izrisovalnik ustvari fizikalno korektne slike, ki natančno upoštevajo naravne pojave kot so npr. lom svetlobe pri prizmi, interferenco in difrakcijo svetlobe.



Slika 5.8: CIE 1931 funkcije za ujemanje RGB barv z valovnimi dolžinami vidnega spektra. (vir: [23]).

Pomanjkljivosti modela RGB odpravi barvni prostor CIE 1931 XYZ. Z modelom XYZ, ki je zasnovan na človeškem vidu, lahko predstavimo vse vidne valovne dolžine. Izrisovalnik interno izrisuje sliko v XYZ prostoru, ko konča, pa jo pretvori v RGB (pri tem pride do izgube natančnosti, vendar zgolj pri pretvorbi končne slike, ne pa med nastajanjem le-te).

Opis spektrov, ki jih uporablja izrisovalnik, sestoji iz porazdelitve moči posameznih valovnih dolžin. Spekter je diskretiziran na vzorce med začetno in končno valovno dolžino (v nanometrih), vzorci pa so med seboj oddaljeni nekaj nanometrov. Vrednost posameznega vzorca je pri objektih, ki ne oddajajo svetlobe (t.j. jo zgolj odbijajo), med 0 in 1 (kjer je 1 popolna odbojnost). Pri objektih, ki oddajajo svetlobo (npr. žarnice, sonce itd.), pa vrednosti vzorcev lahko presegajo 1 (večje vrednosti predstavljajo močnejše izsevnosti pri ustrezni valovni dolžini).

Format datoteke z opisom spektra je preprost. Datoteka je tekstovna in razdeljena na vrstice. V prvi vrstici so tri cela števila, ki določajo: začetno valovno dolžino ($\lambda_{\text{začetna}}$, v nanometrih), končno valovno dolžino ($\lambda_{\text{končna}}$, v nanometrih) in pogostost vzorčenja (torej, koliko nanometrov narazen so vzorci, δ). Tej vrstici sledi $\frac{\lambda_{\text{končna}} - \lambda_{\text{začetna}}}{\delta} + 1$ vzorcev v decimalnem zapisu, vsak vzorec je v svoji vrstici.

5.2.5 BRDFMaterial

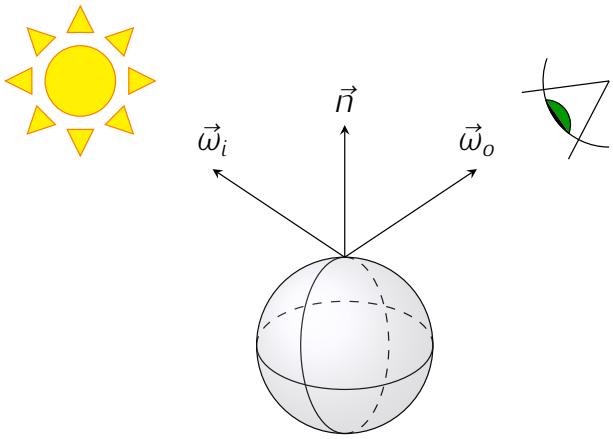
Z operacijo `BRDFMaterial` ustvarimo opis materiala za objekt na podlagi podanega modela BRDF.

BRDFMaterial

V/I	Ime vrat	Tip	Opis
V	`brdf`	`brdf`	Model BRDF
V	`spectrum`	`spectrum`	Barvni spekter
I	`material`	`material`	Opis materiala

Izrisovalnik simulira pot svetlobe po 3D sceni, zato mora vedeti, kako se le-ta odbija od objektov (oz. lomi v primeru prosojnih objektov). BRDF (ang. *bidirectional reflectance distribution function*) je funkcija, ki opisuje odboj svetlobe od neprosojnega objekta [18, 36]. Z njo je definiran izgled materiala (torej, ali izgleda bolj plastično ali bolj kovinsko ipd.). Funkcija ima dva parametra — smer proti luči ($\vec{\omega}_i$) in smer proti kameri ($\vec{\omega}_o$), kot je prikazano na sliki 5.9. Na podlagi teh smeri in pa internih parametrov modela se izračuna odbojnost objekta v točki, ki jo zadane svetloba.

Obstajajo izotropni in anizotropni modeli BRDF [36]. Izotropni imajo isto odbojnost, ne glede na rotacijo okoli vektorja normale objekta ($\vec{n}$). Tako lastnost imajo zelo gladki plastični materiali. Pri anizotropnih pa se odbojnost spreminja glede na rotacijo. Primera materialov s tako lastnostjo sta brušena kovina in lasje.



Slika 5.9: Prikaz vektorjev pri BRDF.

Za fizikalno korektne rezultate mora BRDF imeti dve lastnosti — recipročnost in ohranitev energije [36]. Recipročnost pomeni, da BRDF vrne isto vrednost, če zamenjamo položaja kamere in luči. Ohranitev energije pomeni, da mora biti količina svetlobe, odbite od objekta, manjša ali enaka količini vpadne svetlobe. Torej, BRDF ne sme ustvarjati svetlobe.

Poleg modela BRDF ima opis materiala še barvni spekter, ki določa njegovo barvo. Podrobnejši opis spektrov je v podpoglavju LoadSpectrum.

5.2.6 LambertianBRDF

LambertianBRDF			
V/I	Ime vrat	Tip	Opis
I	`brdf`	`brdf`	Model BRDF

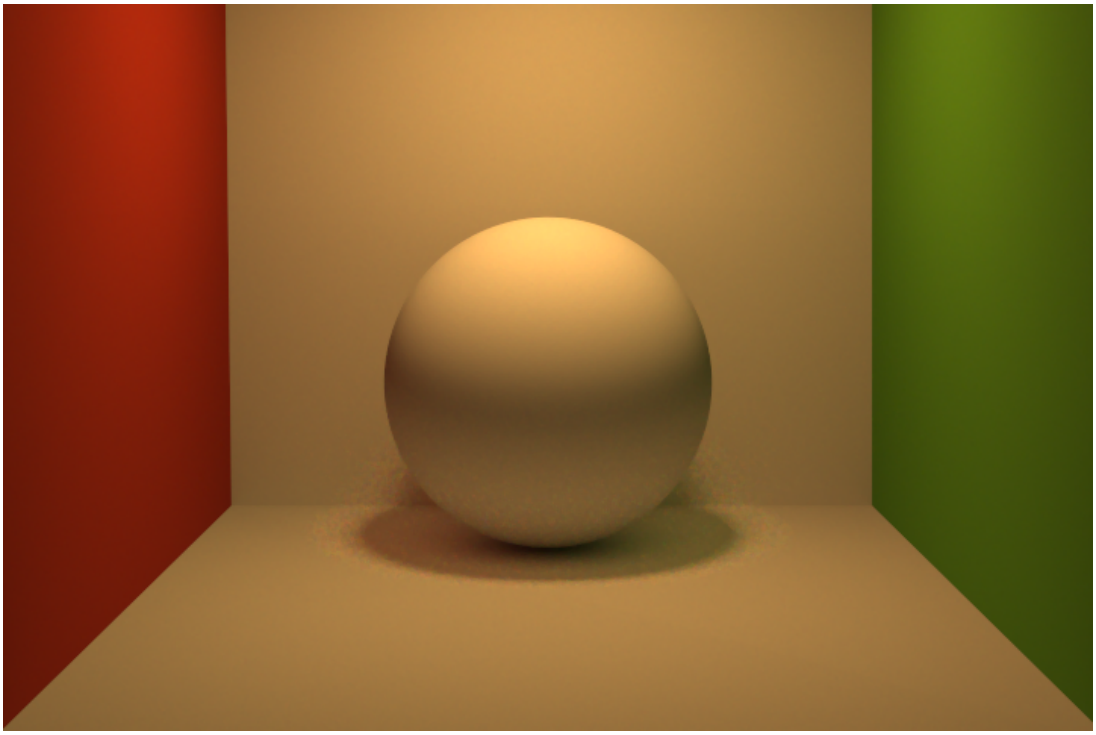
`LambertianBRDF` ustvari Lambertov model BRDF.

Model je izotropen in opisuje idealne difuzne površine, ki v naravi ne obstajajo. Najbližja naravna materiala, ki imata podobne lastnosti, sta neobdelan les in navaden papir (vendar ne, če ga gledamo od strani, ker je takrat videti bleščeč).

BRDF tega modela je izjemno preprosta [18]:

$$\text{BRDF}\,(\vec{\omega}_i, \vec{\omega}_o) = \frac{\rho}{\pi}, \qquad [5.10]$$

kjer je ρ odbojnost točke, ki jo je zadela svetloba.



Slika 5.10: Prikaz Lambertovega modela BRDF.

Slika 5.10 prikazuje kroglo in stene z Lambertovim modelom. Sliko je naredil izrisovalnik, implementiran v tej diplomski nalogi.

5.2.7 AshikhminBRDF

`AshikhminBRDF` ustvari Ashikhminov model BRDF.

AshikhminBRDF

V/I	Ime vrat	Tip	Opis
V	`kd`	`real`	Difuzni koeficient
V	`ks`	`real`	Sijajni koeficient
V	`nu`	`real`	Parameter n_u
V	`nv`	`real`	Parameter n_v
I	`brdf`	`brdf`	Model BRDF

Model je anizotropen in opisuje vrsto naravnih površin [2]. Pri kovinah je difuzni koeficient enak 0, pri poliranih površinah (npr. plastiki) sta oba koeficienta neničelna. S tem modelom lahko simuliramo tudi papir, saj lahko s sijajnim koeficientom opišemo sijaj, ko gledamo list pod velikim kotom. Parametra n_u in n_v določata površino sijaja.

BRDF tega modela je bolj zapletena. V grobem je sestavljena iz sijajne in difuzne komponente:

$$\mathrm{BRDF}\left(\vec{\omega}_i, \vec{\omega}_o\right) = \rho_s\left(\vec{\omega}_i, \vec{\omega}_o\right) + \rho_d\left(\vec{\omega}_i, \vec{\omega}_o\right). \qquad [5.11]$$

Sijajna komponenta je definirana tako:

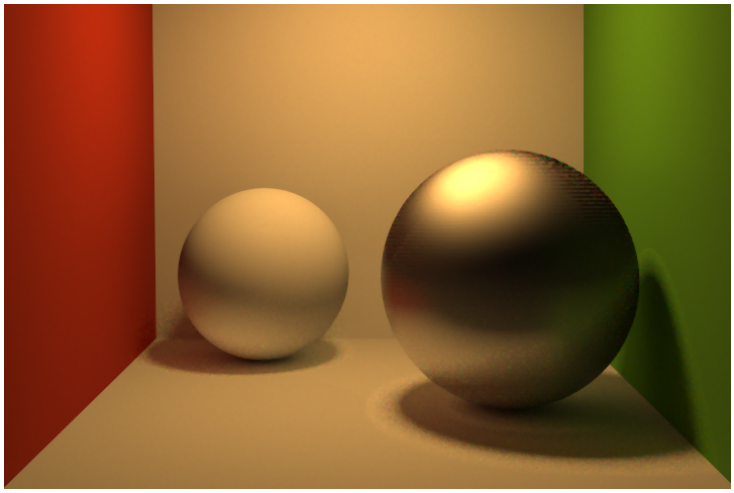
$$\rho_s\left(\vec{\omega}_i, \vec{\omega}_o\right) = \frac{\sqrt{(n_u+1)(n_v+1)}}{8\pi} \frac{\langle \vec{n}, \vec{h}\rangle \frac{n_u\langle \vec{h}, \vec{u}\rangle^2 + n_v\langle \vec{h}, \vec{v}\rangle^2}{1-\langle \vec{h}, \vec{n}\rangle^2}}{\langle \vec{\omega}, \vec{h}\rangle \max\left\{\langle \vec{n}, \vec{\omega_i}\rangle, \langle \vec{n}, \vec{\omega_o}\rangle\right\}} \mathrm{F}\left(\langle \vec{\omega}, \vec{h}\rangle\right). \qquad [5.12]$$

V zgornji enačbi je $\vec{n}$ normala površine, $\vec{u}$ in $\vec{v}$ pa tangentna vektorja nanjo. Skupaj tvorijo ortonormirano bazo (vektorji so si med seboj pravokotni in imajo dolžino 1). Vektor $\vec{h}$ je enak $\frac{\vec{\omega_i}+\vec{\omega_o}}{|\vec{\omega_i}+\vec{\omega_o}|}$. Funkcija $\mathrm{F}(\cos\theta)$ je Fresnelov člen, ki izračuna odbojnost glede na podan kot θ. V članku so uporabili Schlickov približek zanj: $\mathrm{F}(\cos\theta) = k_s + (1-k_s)(1-\cos\theta)^5$.

Difuzno komponento se izračuna po formuli:

$$\rho_d\left(\vec{\omega}_i, \vec{\omega}_o\right) = \frac{28k_d(1-k_s)}{23\pi}\left(1-\left(1-\tfrac{1}{2}\langle \vec{n}, \vec{\omega_i}\rangle\right)^5\right)\left(1-\left(1-\tfrac{1}{2}\langle \vec{n}, \vec{\omega_o}\rangle\right)^5\right). \qquad [5.13]$$

Na sliki 5.11 sta prikazani dve krogli z Ashikhminovim modelom. Krogla v ozadju je skoraj popolnoma difuzna ($k_d = 0.99$, $k_s = 0.01$, $n_u = n_v = 1$), krogla v ospredju pa je iz matirane kovine ($k_d = 0.05$, $k_s = 0.95$, $n_u = n_v = 10$). Stene so iz Lambertovega materiala. Sliko je naredil izrisovalnik, implementiran v tej diplomski nalogi.



Slika 5.11: Prikaz Ashikhminovega modela BRDF.

5.2.8 ApplyEmission

ApplyEmission

V/I	Ime vrat	Tip	Opis
V	`materialIn`	`material`	Vhodni material
V	`emission`	`spectrum`	Izsevni spekter
I	`materialOut`	`material`	Izhodni material

Z `ApplyEmission` materialu dodamo izsevni spekter, kar pomeni, da material začne oddajati svetlobo.

Pri izrisovanju scen mora vsaj en objekt imeti material, ki seva svetlobo, drugače bo končna slika popolnoma črna.

5.2.9 ApplyNormalMap

ApplyNormalMap

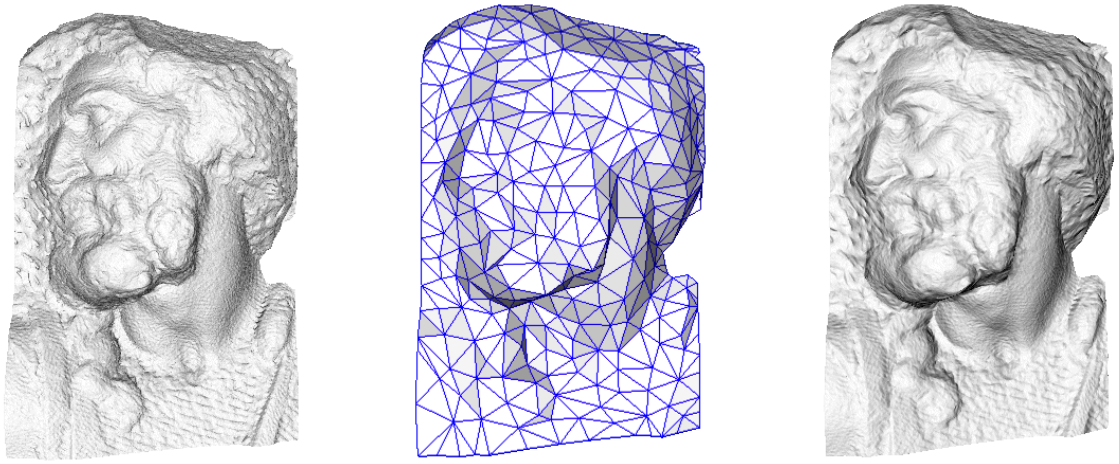
V/I	Ime vrat	Tip	Opis
V	`materialIn`	`material`	Vhodni material
V	`image`	`image`	Mapa normal
I	`materialOut`	`material`	Izhodni material

`ApplyNormalMap` podanemu materialu doda mapo normal.

Mapiranje normal je postopek, s katerim lahko 3D modelu dodamo podrobnosti, ne da bi pri tem uporabili več poligonov (zato je izrisovanje hitrejše).

Koncept je podoben kot pri teksturah, le da se tukaj namesto barve posamezne točke na objektu spremenijo koordinate normale površja objekta. V sliki (mapi normal) predstavljajo RGB komponente posamezne točke odmik X, Y in Z koordinat normale.

Na sliki 5.12 je prikazan primer mapiranja normal.



(a) Originalni model s 4000000 trikotnikov.

(b) Poenostavljen model s 500 trikotniki.

(c) Poenostavljen model z mapiranjem normal.

Slika 5.12: Primer mapiranja normal (vir: [31]).

5.2.10 SceneGraphNode

`SceneGraphNode` ustvari novo vozlišče scenskega grafa. Ustvarjeno vozlišče vsebuje podan objekt in njegovo transformacijsko matriko.

SceneGraphNode

V/I	Ime vrat	Tip	Opis
V	`object`	`obj`	Predmet
V	`transform`	`mat4`	Transformacijska matrika
I	`sgNode`	`sgnode`	Vozlišče scenskega grafa

Scenski graf je hierarhična podatkovna struktura. Vsako vozlišče lahko vsebuje enega ali več objektov, ki si delijo transformacijsko matriko. Poleg tega ima lahko vsako vozlišče še otroke. Transformacije staršev se poznajo tudi na otrocih.

Namen te strukture je, da olajša transformiranje skupin objektov. Če želimo premakniti skupino objektov, ki se nahaja v poddrevesu nekega vozlišča, lahko popravimo zgolj transformacijsko matriko tega vozlišča.

5.2.11 AddChild

Operacija `AddChild` naredi kopijo podanega vozlišča scenskega grafa in ji doda otroka.

AddChild

V/I	Ime vrat	Tip	Opis
V	`parent`	`sgnode`	Prednik
V	`child`	`sgnode`	Otrok
I	`sgNode`	`sgnode`	Vozlišče scenskega grafa

5.2.12 PathTrace

PathTrace

V/I	Ime vrat	Tip	Opis
V	sceneGraph	sgnode	Graf scene
V	camera	camera	Kamera
V	width	int	Širina slike
V	height	int	Višina slike
V	samples	int	Število vzorcev
I	imageOut	image	Izrisana slika

`PathTrace` izriše podano sceno z vidika kamere na sliko podane velikosti. To naredi z algoritmom sledenja poti žarkov (ang. *path tracing*).

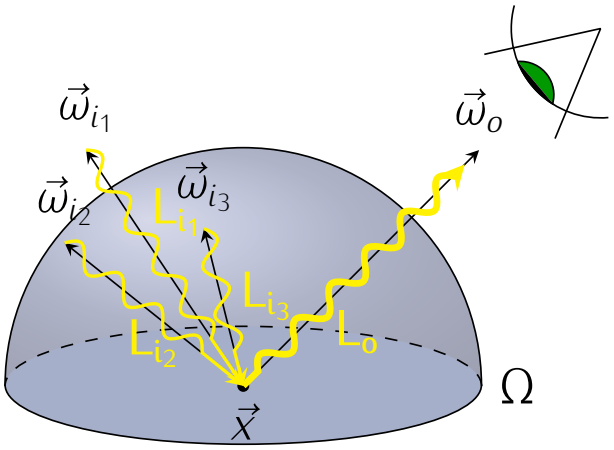
Svetilnost posamezne točke v 3D prostoru lahko izrazimo z enačbo globalnega osvetljevanja [18]:

$$L_o(\vec{x}, \vec{\omega}_o, \lambda, t) = L_e(\vec{x}, \vec{\omega}_o, \lambda, t) + \int_\Omega \mathrm{BRDF}(\vec{x}, \vec{\omega}_i, \vec{\omega}_o, \lambda, t) \cdot L_i(\vec{x}, \vec{\omega}_i, \lambda, t) \cdot \langle -\vec{\omega}_i, \vec{n} \rangle \, \mathrm{d}\vec{\omega}_i. \quad [5.14]$$

Na kratko je pomen te enačbe sledeč:

$$\boxed{\text{izhodna svetilnost}} = \boxed{\text{izsevnost objekta}} + \boxed{\text{prispevki svetilnosti iz vseh možnih smeri}}. \quad [5.15]$$

Izhodna svetilnost L_o, usmerjena iz točke $\vec{x}$ v smeri $\vec{\omega}_o$ v času t pri valovni dolžini λ, je enaka vsoti svetilnosti, ki jo objekt seva sam (L_e), in prispevka svetilnosti iz vseh možnih vpadnih smeri $\vec{\omega}_i$ (glej sliko 5.13).



Slika 5.13: Prikaz enačbe globalnega osvetljevanja.

Prispevek svetilnosti je seštevek svetilnosti (L_i), ki jih prispevajo vsi vpadni žarki svetlobe s področja polkrogle Ω (ta je napeta v smeri normale $\vec{n}$ iz točke $\vec{x}$ na objektu). Pri tem se vsak prispevek duši glede na njegov vpadni kot in BRDF materiala objekta.

Poleg BRDF obstajata tudi BTDF (ang. *bidirectional transmittance distribution function*) in BSSRDF (ang. *bidirectional surface scattering reflectance distribution function*) [14]. BTDF opisuje, kako svetloba potuje skozi prosojen objekt (npr. steklo), BSSRDF pa, kako se svetloba odbija pod površjem nekega objekta, preden ga zapusti (primer materiala s tako lastnostjo je človeška koža).

Ta enačba nima analitične rešitve, zato se je moramo lotiti numerično (z Monte Carlo metodo).

Algoritem sledenja poti žarkov strelja žarke iz kamere v sceno in jih odbija po objektih v njej, pri tem računa svetilnost zadetih točk. Pri računanju prispevkov vzame zgolj naključen

vzorec smeri, saj bi bilo nemogoče pregledati čisto vse. Tak pristop dokazano konvergira k pravi rešitvi [18]. Za vsako točko končne slike opravimo več vzorčenj (parameter `samples`). Več vzorcev da lepše slike (z manj šuma), vendar podaljša čas izrisovanja.

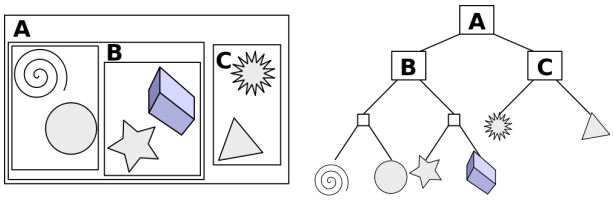
V splošnem algoritem deluje tako:

- za vsako točko izhodne slike:
  - naključno vzorči spekter valovnih dolžin med 400 in 700 nanometri,
  - za vsak vzorec valovne dolžine:
    - izračunaj smer žarka iz kamere proti sceni skozi dano točko, ki jo naključno zamakni,
    - če žarek zadane kak objekt:
      - izračunaj svetilnost v tisti točki,
  - barva točke je povprečje vseh zajetih vzorcev.

Svetilnost in valovno dolžino lahko pretvorimo v XYZ barvo tako, da posamezno komponento nastavimo na svetilnost, uteženo z odzivom ustreznega receptorja v človeških očeh za dotično valovno dolžino.

Pri računanju svetilnosti v zadeti točki pride do nadaljnjih odbojev žarkov.

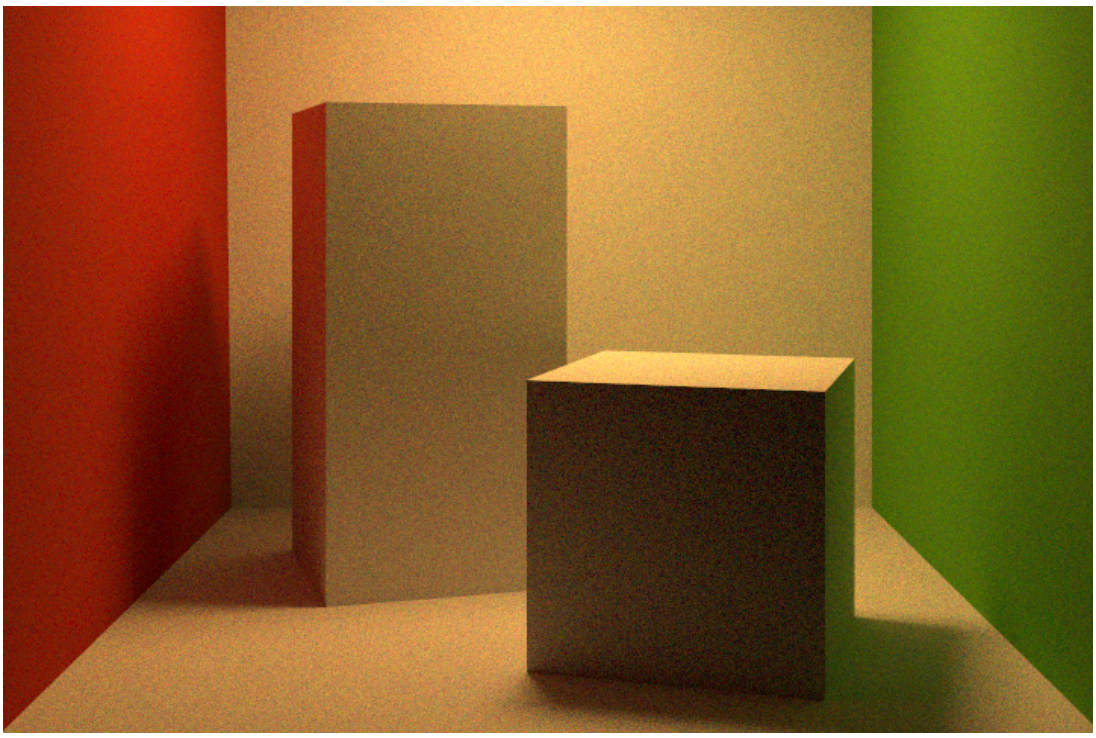
Iskanje presečišč žarka s čisto vsemi objekti v sceni je prepočasno. Računamo lahko le presečišča žarka z objekti, ki so v njegovi bližini. Za to uporabimo strukture za prostorsko indeksiranje, ki razdelijo prostor scene na več delov.



Slika 5.14: Prikaz strukture BVH (vir: [25]).

Izrisovalnik, razvit v tej diplomski nalogi, za prostorsko indeksiranje uporablja strukturo BVH (ang. *bounding volume hierarchy*). Ta razdeli prostor glede na omejitvene prostore posameznih objektov. Prostor, ki ga zaseda posamezen objekt, lahko omejimo s kvadrom. V tem primeru pravimo, da kvader omejuje objekt. Struktura BVH vsebuje drevesno hierarhijo omejitvenih prostorov. Objekti, ki so fizično v istem omejitvenem kvadru, si bodo delili enakega prednika v drevesu (glej sliko 5.14). Razlika med grafom scene in strukturo BVH je ta, da objekti v isti veji grafa scene niso nujno tudi fizično blizu.

Algoritem sledenja poti žarkov se da zelo enostavno povzporediti. Vsako vrstico končne slike lahko izrisujemo na svojem procesorskem jedru.



Slika 5.15: Preizkusna scena za izrisovalnik.

Na sliki 5.15 je prikazan t.i. *Cornell box* [13]. Ta scena se pogosto uporablja za preizkušanje kvalitete izrisovalnikov, ki temeljijo na algoritmu sledenja poti žarkov. Na kvadru in kocki lahko vidimo rdečo in zeleno svetlobo, ki se odbija od sten na objekt in tla. Šum v sliki je posledica Monte Carlo metode — če bi povečali število vzorcev, bi se količina šuma zmanjšala, čas izrisovanja pa povečal.

5.3 Animacija

Trenutno mora uporabnik animacijo izvajati z lastno ustvarjenimi operacijami.

5.4 Splošne operacije

5.4.1 LoadImage

`LoadImage` naloži sliko iz datoteke. Podpira vse formate, za katere obstajajo ustrezni vhodno-izhodni moduli. Trenutno je to zgolj PNG.

LoadImage

V/I	Ime vrat	Tip	Opis
V	`fileName`	`string`	Ime datoteke
I	`imageOut`	`image`	Slika

5.4.2 SaveImage

`SaveImage` shrani sliko v datoteko. S podporo formatov je enako kot pri `LoadImage`.

SaveImage

V/I	Ime vrat	Tip	Opis
V	`fileName`	`string`	Ime datoteke
V	`imageIn`	`image`	Slika

5.4.3 LoadFrame

`LoadFrame` naloži sličico iz video datoteke oz. zaporedja slik. Katero sličico naloži, je odvisno od parametra *t* (tega operaciji poda izvajalnik). Podpira formate, za katere obstajajo vhodno-izhodni moduli. Trenutno je to le zaporedje navadnih slik.

LoadFrame

V/I	Ime vrat	Tip	Opis
V	`fileName`	`string`	Ime datoteke
I	`imageOut`	`image`	Slika

5.4.4 SaveFrame

`SaveFrame` shrani sliko v video datoteko na sličico, ki jo določa parameter *t*.

SaveFrame

V/I	Ime vrat	Tip	Opis
V	`fileName`	`string`	Ime datoteke
V	`imageIn`	`image`	Slika

POGLAVJE 6

Zaključek

Glavni cilj te diplomske naloge je bil ustvariti odprtokodno orodje za opravljanje poljubnih operacij nad podatki z uporabo podatkovno-pretočnega grafa.

V ta namen je bil v diplomski nalogi razvit sistem za vzporedno izvajanje podatkovno-pretočnih grafov z namenskim programskim jezikom za implementacijo operacij v grafih. Kot primer uporabe sistema so bile razvite operacije za kompozitiranje in izrisovanje 3D scen.

Napisanih je bilo več kot 5000 vrstic C++ kode (prešteto z orodjem *c_count*, ki ne upošteva praznih vrstic in vrstic s komentarji). Diplomska naloga je napisana v jeziku LaTeX, diagrami so narejeni vektorsko s knjižnico *TikZ*. Izvorna koda tega dokumenta ima okoli 2500 vrstic.

Implementirani sistem deluje zadovoljivo in izpolnjuje zastavljene cilje, hkrati pa odpira mnogo priložnosti za izboljšave.

Najbolj vidna izboljšava bi bila izdelava grafičnega vmesnika za ustvarjanje grafov. Ta bi zelo olajšal delo in povečal praktično uporabnost sistema.

Sistem bi se lahko razširilo s podporo za porazdeljeno izvajanje grafov na več računalnikih. Razporejevalnik bi lahko upošteval tudi težavnost posameznih operacij in lokalnost podatkov (opravila bi poskušal dodeliti tistim procesorskim jedrom, ki imajo potrebne podatke že v predpomnilniku). Implementacija prioritetne vrste v razporejevalniku trenutno uporablja zaklepanje. Če bi se jo zamenjalo s tako implementacijo, ki zaklepanja ne potrebuje, bi bili dostopi do te strukture hitrejši. Vse to bi izboljšalo hitrost izvajanja grafov.

Jeziku bi lahko dodali vzporedno varianto `for` zanke, ki bi uporabljala isti razporejevalnik opravil. Če bi sintakso in semantiko te zanke malce spremenili, da bi bila bolj podobna tisti v Pascalu, bi lahko na enostaven način izvajali tudi avtomatsko povzporejanje zank.

Uporabo jezika bi olajšal operator za dostop do elementov struktur, tako kot to lahko počnemo s piko v jeziku C. Da bi ohranili čistost jezika, ta operator ne bi smel dopuščati pisanja v elemente struktur — pri spreminjanju struktur je bolj smiselno narediti novo kopijo strukture s popravljenimi elementi.

Dodatna možnost za pohitritev izvajanja jezika je v implementaciji določene optimizacije za LLVM. LLVM zna pretvoriti klice funkcije `malloc()`, ki pridobi pomnilnik, v rezerviranje prostora na skladu, če ugotovi, da se pridobljen pomnilnik uporablja zgolj znotraj določene funkcije. Ker pa razviti jezik namesto funkcije `malloc()` uporablja `GC_malloc()`, LLVM tega ne zazna in ne more opraviti te optimizacije.

Moč sistema se lahko poveča z dodajanjem novih operacij. Na področju računalniške grafike je še veliko zanimivih operacij, ki bi lahko bile del tega sistema. Trenutno manjkajo operacije, ki bi olajšale animacijo. Možne so tudi izboljšave že obstoječih operacij.

Implementirali bi lahko tudi operacije s področij procesiranja zvoka in signalov, računalniškega vida, podatkovnega rudarjenja in upravljanja strojne opreme. S takimi operacijami bi lahko naredili graf, ki zajema meritve s senzorjev, dela nad njimi analize in rezultate vizualizira. S sistemom bi lahko tudi krmilili robota.

Slike

Literatura

[1] Apple. *Shake 4 Tutorials*. 2005. Dostopno na: http://manuals.info.apple.com/en/Shake_4_Tutorials.pdf. Zadnji dostop: 30. maj 2013.

[2] Michael Ashikhmin in Peter Shirley. An anisotropic Phong BRDF model. Dostopno na: http://www.cs.utah.edu/~michael/brdfs/jgtbrdf.pdf. Zadnji dostop: 7. junij 2013.

[3] J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden in M. Woodger. Revised report on the algorithm language ALGOL 60. *Communications of the ACM*, 6(1):1–17, januar 1963.

[4] Ron Brinkmann. *The art and science of digital compositing*. Morgan Kaufmann Publishers Inc., 1999.

[5] Edward Cannon. Chromakey algorithm. Dostopno na: http://gc-films.com/chromakey.html. Zadnji dostop: 6. junij 2013.

[6] Bill Desowitz. Returning to Middle-Earth with *The Hobbit*. Dostopno na: http://www.awn.com/print/articles/visual-effects/returning-to-middle-earth-with-the-hobbit?page=0%2C0. Zadnji dostop: 16. april 2013.

[7] William Fetter. Boeing man. Dostopno na: http://siudesign.org/index_htm_files/228.png. Zadnji dostop: 7. maj 2013.

[8] Blender Foundation. OpenGL render of the hand rig of the main character in the open source project Sintel. Dostopno na: http://upload.wikimedia.org/wikipedia/commons/f/f2/Sintel-hand.png. Zadnji dostop: 23. maj 2013.

[9] Side Effects Software Inc. Houdini 9.5: Compositing: Blend or layer images. Dostopno na: http://www.sidefx.com/docs/houdini9.5/composite/layers. Zadnji dostop: 5. junij 2013.

[10] Mahmut Kandemir, N. Vijaykrishnan, Mary Jane Irwin in Wu Ye. Influence of compiler optimizations on system power. *IEEE Transactions on VLSI Systems*, 9(6):801–804, december 2001.

[11] George A. Michael. Early computer graphics work and equipment at LLNL. Dostopno na: http://www.computer-history.info/Page4.dir/pages/Early.Graphics.dir/. Zadnji dostop: 4. maj 2013.

[12] The University of Auckland. Geometric operations. Dostopno na: http://www.cs.auckland.ac.nz/courses/compsci773s1c/lectures/ImageProcessing-html/topic2.htm. Zadnji dostop: 6. junij 2013.

[13] Cornell University Program of Computer Graphics. Cornell box data. Dostopno na: http://www.graphics.cornell.edu/online/box/data.html. Zadnji dostop: 9. junij 2013.

[14] Matt Pharr in Greg Humphreys. *Physically Based Rendering, Second Edition: From Theory To Implementation*. Morgan Kaufmann Publishers Inc., 2010.

[15] Thomas Porter in Tom Duff. Compositing digital images. *SIGGRAPH Computer Graphics*, 18(3):253–259, januar 1984.

[16] The LLVM Project. Dostopno na: http://www.llvm.org/. Zadnji dostop: 30. april 2013.

[17] Martin Richards in Colin Whitby-Stevens. *BCPL — the language and its compiler*. Cambridge University Press, 1979.

[18] Peter Shirley in R. Keith Morley. *Realistic Ray Tracing, 2nd edition*. A. K. Peters, Ltd., 2003.

[19] Abraham Silberschatz, Peter Baer Galvin in Greg Gagne. *Operating System Concepts, 8th edition*. Wiley Publishing, 2008.

[20] Greg Ward. A contrast-based scalefactor for luminance display. 1994. Dostopno na: http://gaia.lbl.gov/btech/papers/35252.pdf. Zadnji dostop: 4. junij 2013.

[21] Wikipedia. Apple Cinema Display. Dostopno na: http://en.wikipedia.org/wiki/Apple_Cinema_Display. Zadnji dostop: 4. junij 2013.

[22] Wikipedia. CbCr plane of the YCbCr color space. Dostopno na: http://upload.wikimedia.org/wikipedia/commons/thumb/3/34/YCbCr-CbCr_Scaled_Y50.png/768px-YCbCr-CbCr_Scaled_Y50.png. Zadnji dostop: 6. junij 2013.

[23] Wikipedia. CIE 1931 RGB CMF. Dostopno na: http://upload.wikimedia.org/wikipedia/commons/thumb/6/69/CIE1931_RGBCMF.svg/1000px-CIE1931_RGBCMF.svg.png. Zadnji dostop: 5. junij 2013.

[24] Wikipedia. Combination printing. Dostopno na: http://en.wikipedia.org/wiki/Combination_printing. Zadnji dostop: 4. maj 2013.

[25] Wikipedia. Example of bounding volume hierarchy. Dostopno na: http://upload.wikimedia.org/wikipedia/commons/thumb/2/2a/Example_of_bounding_volume_hierarchy.svg/1000px-Example_of_bounding_volume_hierarchy.svg.png. Zadnji dostop: 9. junij 2013.

[26] Wikipedia. History of computer animation. Dostopno na: http://en.wikipedia.org/wiki/History_of_computer_animation. Zadnji dostop: 7. maj 2013.

[27] Wikipedia. History of programming languages. Dostopno na: http://en.wikipedia.org/wiki/History_of_programming_languages. Zadnji dostop: 25. maj 2013.

[28] Wikipedia. Houdini (software). Dostopno na: http://en.wikipedia.org/wiki/Houdini_(software). Zadnji dostop: 16. april 2013.

[29] Wikipedia. Jacquard loom. Dostopno na: http://en.wikipedia.org/wiki/Jacquard_loom. Zadnji dostop: 23. maj 2013.

[30] Wikipedia. Luminance (relative). Dostopno na: http://en.wikipedia.org/wiki/Luminance_(relative). Zadnji dostop: 4. junij 2013.

[31] Wikipedia. Normal map example. Dostopno na: http://upload.wikimedia.org/wikipedia/commons/3/36/Normal_map_example.png. Zadnji dostop: 6. junij 2013.

[32] Wikipedia. Programming paradigm. Dostopno na: http://en.wikipedia.org/wiki/Programming_paradigm. Zadnji dostop: 10. junij 2013.

[33] Wikipedia. Wavefront .obj file. Dostopno na: http://en.wikipedia.org/wiki/Wavefront_.obj_file. Zadnji dostop: 5. junij 2013.

[34] Wikipedia. Zoetrope. Dostopno na: http://en.wikipedia.org/wiki/Zoetrope. Zadnji dostop: 5. maj 2013.

[35] Niklaus Wirth. *Compiler construction*. Addison-Wesley, 1996. Dostopno tudi na: http://www.ethoberon.ethz.ch/WirthPubl/CBEAll.pdf. Zadnji dostop: 25. maj 2013.

[36] Chris Wynn. An introduction to BRDF-based lighting. Dostopno na: http://www.cs.ucla.edu/~zhu/tutorial/An%20Introduction%20to%20BRDF-Based%20Lighting.pdf. Zadnji dostop: 7. junij 2013.