Tehnike rasterizacije

Raguzin, Mauro

Sveučilište u Rijeci – Odjel za informatiku

Preddiplomski studij informatike

Mauro Raguzin

Tehnike rasterizacije

Završni rad

Mentor: dr. sc. Vedran Miletić
Komentor: dr. sc. Sanda Bujačić Babić

Rijeka, 19. srpnja 2020.

Sadržaj

Sažetak

U ovom završnom radu bit će obrađena tema računalne grafike s fokusom na trodimenzionalno renderiranje. Pritom će se detaljno objasniti uloga današnjih GPU-ova u sustavima za 3D renderiranje te objasniti kako točno neki od poznatijih algoritama rade. Fokus je prvo na danas još uvijek najvažnijem osnovnom pristupu 3D renderiranju – rasterizaciji – a potom ćemo se vratiti i na raytracing kao zanimljivu, atraktivnu no procesorski mnogo zahtijevniju tehniku. Potonja također dobiva i svoju GPU akceleriranu implementaciju u AMD ROCm API-ju kao dio ovog rada, zajedno s popratnom detaljnom dokumentacijom i izvornim kodom (C++).

Pretpostavlja se osnovno poznavanje pojmova linearne algebre i numeričke matematike, za što bi čitatelju bilo dovoljno proučiti prijašnji rad [1]. Nagalasak je u ovom radu stavljen na *osnovne* algoritme svojstvene svakoj od navedenih tehnika iz čega se može iščitati koja tehnika (rasterizacija) je *efikasnija* od druge te *zbog kojeg* algoritma te tehnike je tome tako. Također, na kraju rada slijedi kratak pregled standardnog grafičkog cjevovoda kakav se može naći implementiran u svakom grafičkom procesoru današnjice.

Ključne riječi

3D renderiranje, rasterizacija, raytracing, ROCm

1 Uvod

Čitanjem ovog teksta na računalnom zaslonu, naše oči simultano primaju informacije od više tisuća sitnih kristala koji emitiraju svjetlost neke od osnovnih triju boja – crvene, zelene i plave. Integriranjem te svjetlosti koja dolazi iz diskretnih, neprimjetno sitnih izvora uzduž relativno velike površine zaslona, naš um stvara percepciju neprekidne, glatke slike, koja u najboljem slučaju može uspješno odavati dojam realizma.

Cilj 3D renderiranja u najnaprednijoj računalnoj grafici je u osnovi to – postići praktički neprimjetnu razliku između onoga što očekujemo vidjeti u našem neposrednom fizičkom trodimenzionalnom okruženju i onoga što nam se prikazuje na dvodimenzionalnom računalnom zaslonu. Često se javljaju i dodatni praktični zahtjevi, poput onih ekonomskih, vezanih za latenciju prikaza te potrošnju električne energije od strane računala i svih njegovih komponenti.

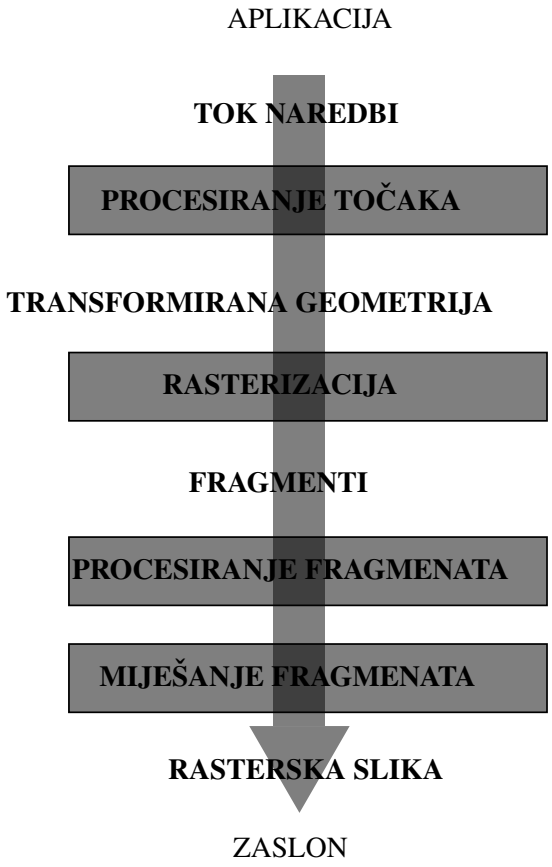
Iako se danas u raznim softverskim poljima koriste različiti napredni algoritmi za 3D renderiranje, gotovo svi oni i dalje ovise o standardiziranim API-jevima koje kreiraju i kontroliraju razna standardizacijska tijela i konzorciji. Ono što možemo vidjeti kao glavnu zajedničku točku svih danas najpopularnijih API-jeva jest da su oni bazirani na *rasterizaciji*, tj. pristupu renderiranju gdje se primitivni oblici – najčešće trokutovi – smješteni u određeni 3D koordinatni sustav te se posebnim nizom transformacija prenose u ekranski sustav korisnikovog računala, gdje se tada i iscrtavaju (renderiraju). Pritom je potrebno primijeniti određene algoritme koji ispunjavaju one regije ekrana koje okupiraju pojedini trokuti, i to na način da se poštuje njihov odnos po dubinskoj osi – pikseli trokutova koji su ispred stražnjih trokutova moraju „pregaziti" sve one stražnje piksele.

Prilikom ispunjavanja trokutova se događa sve ono što prikazanoj sceni daje njen karakter – od konstantnih tonova boja, preko varirajućih detaljnih presijavanja određenih dijelova nekih površina do naprednih efekata poput refrakcije i refleksije okolne slike od dane površine (npr. vode). Najosnovnija vrsta ispune uključuje apliciranje teksture ili boje/boja na dan trokut. Upravo je ovaj okvir rasterizacije – onaj koji podržavaju danas svi GPU-ovi primarno putem pixel shadera – onaj koji obrađujemo detaljno u ovom radu, kroz pregled nekih povijesno zanimljivih algoritama koji su osnova i za mnoge današnje pristupe u složenim algoritmima koji se tipično implementiraju kroz sinergiju mnogih dostupnih shadera, o čemu ćete pročitati u idućem poglavlju.

Tehnike koje nam omogućuju izvođenje spomenutih transformacija i točno vođenje računa o dubini su detaljno objašnjene u [1] (poglavlje 3), pa ih ovdje nećemo ponavljati. Dovoljno je znati i sada pretpostaviti da imamo sve točke naših trokutova smještene u svjetskom prostoru i možemo nastaviti s detaljnim obrađivanjem rasterizacije, a potom i implementacije jednog GPU raytracer-a.

Raytraceri se mogu objasniti na mnogo lakši način od rasterizacije te zahtijevaju mnogo manje prethodne transformacijskih alata radi renderiranja jednostavne scene, no kod ozbiljnijih optimizacija na današnjim GPU sustavima i ovdje dolazimo do zanimljivih i složenih problema, što je i opisano u posljednjem poglavlju ovog rada, a još detaljnije obrazloženo kroz cijeli izvorni kod i dokumentaciju popratnog softverskog rješenja.

No prvo moramo objasniti, radi boljeg razumijevanja konteksta unutar kojeg se sve ovo spominje, kako rade današnji grafički procesori ili GPU-i (engl. *Graphics Processing Unit*). Iduće poglavlje će, prije uvođenja naprednijih pojmova s današnjih arhitektura, radi kontinuiteta ponoviti jedan uvodni pregled grafičkog cjevovoda preuzet iz [1], tako da čitatelj upoznat s tim materijalom može odmah nastaviti sa poglavljem 2.4.



Slika 1. Stadiji grafičkog cjevovoda.

2 Kratki pregled standardnog grafičkog cjevovoda

Objasnili smo kako dovesti točke iz našeg trodimenzionalnog prostora [1] (poglavlje 3) na razne načine u ravninu slike, tj. na samu sliku, no to nije dovoljno kako bi se sve te točke pripadne geometrije zaista i *vidjele* na ekranu. Za to, valja opisati niz operacija koje je potrebno izvesti nakon ovog koraka kako bi se na kraju dobili pikseli slike. Niz koraka koji se izvodi poznat je kao *grafički cjevovod*.

Objektno renderiranje u današnje vrijeme uživa znatan uspjeh, primarno zbog toga što ono počinje od objekata, a završava s pikselima; drugim riječima, kroz objekte scene potrebno je proći samo jedanput kako bi se odgovarajući pikseli obnovili, dok to nije nužno slučaj u slikovnom renderiranju, stoga su performanse objektnog renderiranje jedno od glavnih prednosti ovog pristupa. Ipak, jedinstveni grafički cjevovod za ovakav pristup renderiranju ne postoji — različiti API-jevi poput OpenGL-a i Direct3D-a koriste različite cjevovode od Pixarova RenderMan-a koji je, za razliku od potonjih optimiziranih za rad u realnom vremenu, namijenjen što kvalitetnijim slikama namijenjenim filmovima nauštrb vremenu renderiranja. Naravno, zajednički elementi svih takvih cjevovoda postoje te ćemo upravo te dijelove ovdje ukratko predstaviti.

Sa slike 1 vidimo glavne stadije grafičkog cjevovoda, koji se općenito može organizirati u operacije koje prethode rasterizaciji, sama rasterizacija te operacije nakon rasterizacije. Za operacije koje prethode, dakako, standardno je projicirati sve točke scene na način koji smo opisali u [1], rezultirajući u koordinatama piksela u tzv. *ekranskom* prostoru koji se isključivo koriste u samoj rasterizaciji. Nakon rasterizacije, najčešće se koristi neki algoritam koji eliminira skrivene površine, tj. one koje kamera ne vidi, a popularni odabir za to je *depth buffer*, poznat kao *z-buffer* algoritam.

2.1 Rasterizacija

Rasterizacija je središnja operacija u objektnom renderiranju; za ulazni skup projiciranih točaka, ona *određuje* pripadne ekranske piksele pokrivene tim skupom točaka te *interpolira* vrijednosti nazvane atributima po tim pikselima. Izlaz ove operacije cjevovoda je skup *fragmenata*, jedan za svaki piksel pokriven ulaznim skupom točaka. Svaki fragment „živi" na pojedinom pikselu i nosi vlastiti skup atributa. Ipak, prije nego što se može krenuti u rasterizaciju nekih jednostavnih primitiva poput pravaca i trokuta, mora se obaviti obrezivanje.

2.1.1 Obrezivanje

Naivna rasterizacija projiciranih točaka na ekran ne daje konzistentno dobre rezultate. Naime, objekti koji su izvan kanonskog volumena pogleda — pogotovo oni iza kamere — mogu ipak završiti rasterizirani, što dovodi do netočnih rezultata. Zbog svih tih razloga zaključujemo da je prije same rasterizacije nužno obaviti tzv. *obrezivanje* točaka koje se protežu iza kamere. Najčešće se, međutim, ide na obrezivanje točaka po svih šest kanonskih ravnina piramide pogleda, što je opisano u [1] (poglavlje 3.3.1). Tome se može pristupiti na dva uobičajena načina:

1. U globalnim koordinatama, koristeći šest ravnina koje omeđuju krnju piramidu pogleda.
2. U 4D transformiranom prostoru prije (engl. *clip space*) perspektivnog dijeljenja.

Oba načina se mogu efikasno implementirati; kasnije ćemo vidjeti jednu implementaciju rasterizatora koji ne ovisi o eksplicitnom obrezivaču, već sve obavlja u *clip space* koordinatama koje se u što kasnijoj točki cjevovoda pretvore u konačne piksel koordinate, pri čemu je obrezivanje moguće riješiti u toku skeniranja framebuffera tj. „implicitno".

2.1.2 Crtanje pravaca

Iscrtavanje pravaca kao najjednostavnijih primitiva koje povezuju dvije projicirane točke (nakon obrezivanja) najčešće se izvodi pomoću Bresenhamovog[1] algoritma.

2.1.3 Crtanje trokuta

Trokuti su najčešće korištena primitiva u računalnoj grafici, pa je razumljivo da su algoritmi vezani upravo za njihovu rasterizaciju posebno optimizirani u grafičkom hardveru. Standardan pristup ovom problemu je zadati tri točke u ekranskom prostoru zajedno s odgovarajućim *bojama* tih točaka (vrhovima trokuta) koja se tada linearno interpolira kroz cijelu unutrašnjost trokuta kako bi se dobio osjenčan trokut. Uz dodatne atribute, kao što su koordinate tekstura i ostalih elemenata, mogu se postići realističniji i oku ugodniji rezultati s razumnim porastom kompleksnosti računanja, pa tako i produljenja vremena algoritma. Običaj je pritom koristiti baricentrične koordinate i prikladan Gouraudov[2] tip interpolacije.

2.2 Procesiranje fragmenata

Nakon što smo odredili fragmente pripadnih piksela neke projicirane i obrezane površine, došlo je vrijeme da se svi atributi vezani za te fragmente, poput boje, točno primjene na izlazni piksel koji mora, naravno, dobiti svoju boju i dubinsku vrijednost kako bi se uspješno odredio najbliži piksel kameri. To je zadaća ovog stadija cjevovoda.

[1] Jack Elton Bresenham (1937.), američki računalni znanstvenik.
[2] Henri Gouraud (1944.), francuski računalni znanstvenik.

Ovdje se moraju obaviti svi željeni računi sjenčanja; za najjednostavniji slučaj linearne interpolacije boje od jedne točke do druge, dovoljno je iskoristiti već interpoliranu boju koju nam isporuči stadij rasterizacije (jer on već linearno interpolira trokute, pa možemo iskoristiti, za dani piksel trokuta, sve interpolirane atribute, pa tako i boju). Ipak, za sve složenije modele je potrebno raditi daljnje izračune – u današnjem hardveru, to uglavnom obavljaju zasebne jedinice u grafičkom hardveru koje u tu svrhu pokreću zasebne tzv. *fragment shader* programe. Jedan od jednostavnijih takvih modela je tzv. sjenčanje po vrhovima[3], a jedan od složenijih, ali realističnih, jest sjenčanje po fragmentima[4]. Analogon *fragment shader* programa jesu *vertex shader* programi koji su u prvom stadiju cjevovoda zaduženi za matrično množenje vrhova geometrije scene s matricom željene projekcije u ekranski prostor (najčešće ovi programi ipak samo transformiraju 3D svjetske koordinate u *clip space* NDC koordinate koje zatim standardni rasterizator danog API-ja koristi za iscrtavanje primitiva).

2.3 Miješanje fragmenata

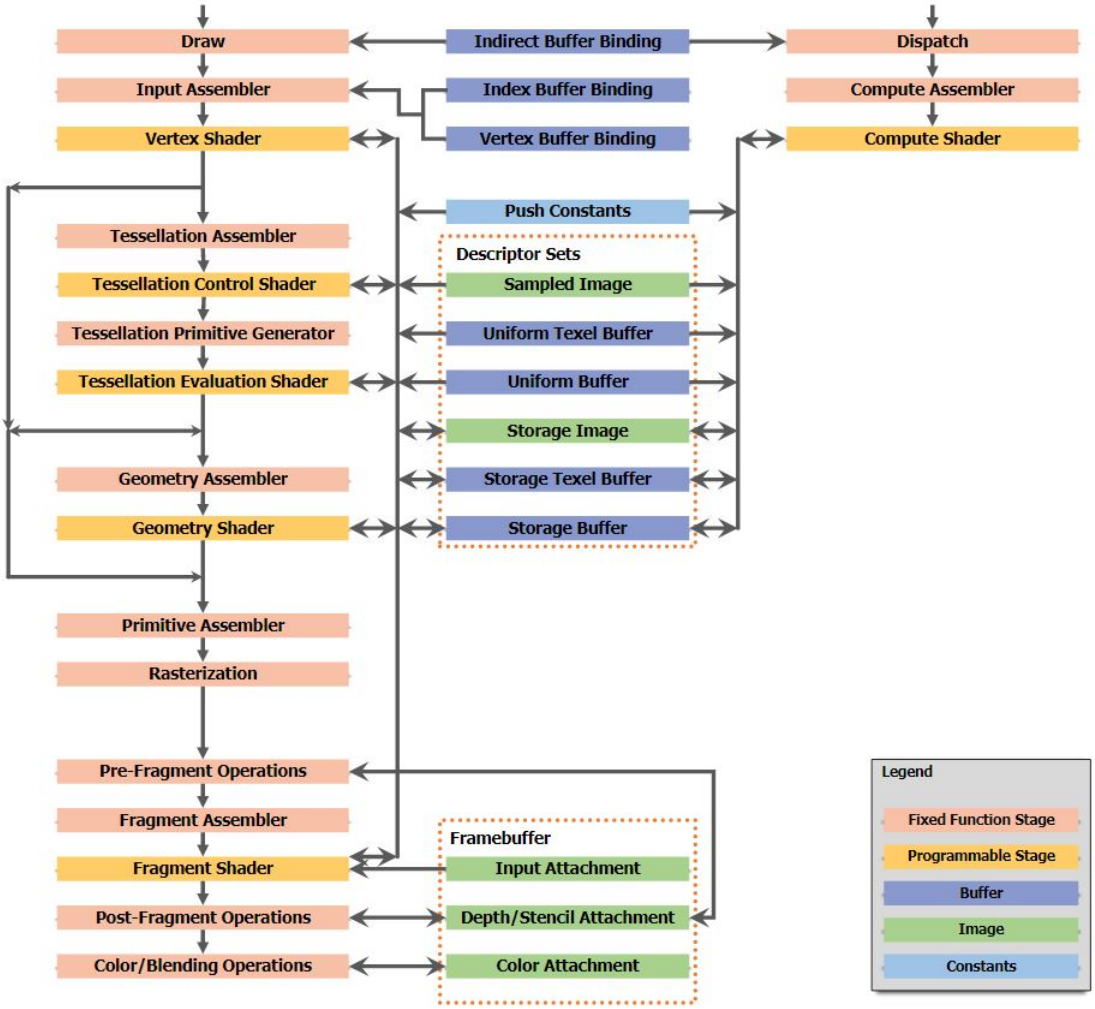
Konačno, svi se generirani fragmenti pojedinih piksela moraju pomiješati u konačne piksele koji čine sliku. To se najčešće čini spomenutim *depth buffer* algoritmom, gdje se odabire onaj fragment s *najmanjom* dubinskom (z) vrijednosti te se on proglasi konačnim pikselom na toj poziciji. Dakako, ako je potrebno iscravati bilo kakve efekte koji uključuju prozirnost objekata, tada *nije* dovoljno koristiti samo z-buffer, već je općenito potrebno sortirati sve poligone u sceni prema njihovoj svjetskoj z-koordinati te ih renderirati u striktnom dalje-bliže uređaju. Tu se mogu javiti problemi oko efikasnosti tog postupka i posebnim degeneriranim slučajevima kod geometrije u sceni koja se preklapa s nekom drugom, što može dovesti do artefakata u renderiranju. Za sve te probleme postoje razna rješenja u koja ovdje nećemo ulaziti jer ovdje implementirani rasterizator očekuje jednostavne scene gdje sortiranje nije potrebno, ili striktno statične scene koje uključuju efikasan hijerarhijski omeđujući volumen (engl. *Bounding volume hierarchy – BVH*) poput BSP-a koji omogućuju trivijalno rješavanje tog problema u realnom vremenu.

2.4 Detaljniji opis modernog grafičkog cjevovoda

U prošloj smo sekciji vidjeli kako općenito funkcionira rasterizacija na najvišoj razini. Sada ćemo pokušati oprimjeriti gornje elemente s jednim modernijim, konkretnim cjevovodom koje koriste današnje grafičke kartice i u kojemu možemo pronaći potonje elemente uz još neke dodatne. Pritom nam neće biti cilj dati sveobuhvatan pregled svih pojmova i koncepata koji se koriste pri modernom grafičkom programiranju pomoću suvremenih API-jeva, već samo orijentacijsko izlaganje onih principa koji su se u posljednjih dvadesetak godina ustoličili pri grafičkom programiranju i većini grafičkih API-jeva.

Na slici 2 vidimo dva glavna odvojena stadija rasterizacijskog cjevovoda, obojana svjetlo narančasto i tamno žuto [3]. Prvi stadij, *Fixed Function Stage*, označava funkcionalnosti GPU-a koje nisu programabilne jer se bave visoko optimiziranim, generičkim operacijama s geometrijom i ostalim objektima koje je korisnik dao API-ju na grafičko procesiranje i nad kojima više ne treba imati nikakvu detaljniju kontrolu pored eventualno namještanja željenih postavki. Te operacije uključuju spajanje svih ulaznih podataka za trenutnu sličicu (u slučaju da radimo klasično vizualno renderiranje, a ne nešto apstraktnije korištenje GPU-a samo za računanje bez potrebe prikazivanja ikakvih rezultantnih slika) u jednu cjelinu (*Input Assembler*) s kojima se tada kasnije barata. Primijetimo da ovaj stadij sadrži i operacije slaganja primitiva (*Primitive Assembler*), najčešće trokutova, kako bi se mogli rasterizirati u idućoj operaciji, kao i operacije nad pojedinim fragmentima koje *prethode* primjeni fragment shadera (*Pre-Fragment Operations* te one koje *slijede* primjenu shadera (*Post-Fragment Operations*. Ta distinkcija omogućava procesiranje fragmenata nezavisno od shader operacija, koje mogu biti vrlo složene, prije nego što ih shader dotakne i nakon što je shader obavio svoj posao nad njima. Prije nego što fragment shader može biti

[3] *engl. per-vertex shading*
[4] *engl. per-fragment shading*



Slika 2. Potpuni cjevovod s modernih GPU-ova, sa svim mogućim prikazanim stadijima.

pokrenut za konretan fragment, različite varijable koje se interpoliraju duž cijele primitive moraju biti spremne za evaluaciju na toj konkretnoj koordinati te ostale varijable koje su invarijantne tijekom cijelog tog procesa moraju također biti dostupne fragment procesoru koji će izvršiti taj shader – sve to i možda više rade pred-fragment operacije koje zato moraju primati podatke koje je vertex shader izračunao (one varijable koje se moraju perspektivno-korektno linearno interpolirati uzduž cijele primitive treba evaluirati na svakom fragmentu, što tipično radi rasterizator). Primijetimo također da post-fragment operacije komuniciraju s dostupnim z-bufferom koji, ako se koristi, omogućuje da se dubina trenutnog fragmenta zapiše na trenutni element z-buffera za kasnije korektno renderiranje, dakako nakon što se prvo pročita stara vrijednost i odluči treba li ili ne onda iscrtati ovaj fragment. Ovaj stadij završava s operacijama miješanja boje trenutnog fragmenta s prethodnim bojama, ako je taj efekt potreban, nakon čega rezultirajuća slika biva poslana u framebuffer, ili točnije u tzv. *Color Attachment* koji sadrži sve piksele framebuffera u željenim postavkama i formatima boje. Ovdje valja napomenuti da se fiksni proces obrezivanja prema ovom cjevovodu, koji ga ne navodi kao zaseban korak u fiksnom stadiju, mora implementirati ili pri slaganju primitiva (eksplicitno) ili pri samoj rasterizaciji (implicitno), nakon što su svi prijašnji koraci generirali sve konačne primitive za scenu.

Drugi glavni stadij, onaj koji je programabilan (*Programmable Stage* na 2), sastoji se od svih koraka koji uključuju izvođenje različitih tipova shader programa. Ti su programi, po dizajnu, pod kontrolom korisnika (programera grafičke aplikacije) koji ih tipično piše u različitim specijaliziranim jezicima više razine, od kojih je danas među najvažnijima (poglavito zbog svoje otvorenosti i dostupnosti) OpenGL Shading Language (GLSL) [4], baziran na pojednostavljenoj i specijaliziranoj C++ sintaksi. Kao i svaki računalni program, svaki shader pisan u takvom jeziku se prvo mora prevesti na konkretni strojni jezik GPU-a kako bi ga pojedini procesori tamo mogli izvoditi; to se prevođenje i konačno povezivanje u prošlosti isključivo odvijalo na klijentskoj strani, tj. CPU-u koji je izvodio pogonske programe (engl. *driver*) GPU-ova prije nego što je bilo potrebno izvoditi dotične programe, što je omogućavalo vrlo jednostavno prenošenje jedanput napisanih programa na vrlo različit hardver, no s druge strane je često dovodilo do neočekivanih usporavanja renderiranja ili različitih vrsta zastajkivanja u gotovo nasumičnim trenucima renderiranja. Trenutno rješenje ovog problema nudi SPIR-V u obliku međukoda (engl. *Intermediate Representation – IR*) za shader programe čije je konačno prevođenje, za razliku od GLSL-ovog, puno efikasnije i jednostavnije za većinu GPU-ova (zbog toga što se tu shaderi ne sastoje od cijelih izvornih kodova shadera, već samo od pretprocesiranih i donekle optimiziranih reprezentacija koje ciljaju minimalno ali dovoljno apstraktan stroj), što i dalje omogućuje visoku prenosivost programa, ali uklanja većinu spomenutih problema s performansama u kritičnim aplikacijama, pogotovo u kombinaciji s novim Vulkan grafičkim API-jem. SPIR-V se može koristiti i uz najnovije verzije OpenGL API-ja. U najopćenitijem i najboljem slučaju, dani GPU može podržavati iduće tipove shader programa: vertex shader, fragment (piksel) shader, shader za kontrolu teselacije, shader za evaluaciju teselacije, geometrijski shader te računski (*compute*) shader. U počecima masivne GPU proizvodnje 1990-ih, svi su komercijalni GPU-ovi (tada poznati kao 3D akceleratori) bili isključivo fiksne funkcionalnosti, tj. nisu podržavali nikakvu vrstu korisničkih programa, već su sve operacije izvodili u sklopu optimiziranog vlasničkog hardvera s algoritmima koji su najviše odgovarali konkretnoj (mikro)arhitekturi i zadovoljavali specifikacije određenih API-jeva (u najranijim danima, nije čak bilo ni nikakvih standardiziranih API-ja na koje se moglo u potpunosti računati da ih svi GPU-ovi podržavaju). Kasnije, prva generacija programabilnih GPU-ova je na tržište donijela osnovne programabilne GPU-ove s vertex i piksel procesorima koji su po prvi puta bili pod kontrolom programera, sa znatnim restrikcijama oko iskoristivih resursa. Danas, skoro svaki diskretni i integrirani (osim nekih starijih mobilnih) GPU nativno podržava sve gore navedene shadere. Ovdje koristimo OpenGL/Vulkan/Khronos nazivlje pojedinih shadera, koji se u drugim API-jevima mogu zvati malo drugačije, mada se uvijek moraju preslikavati na isti skup funkcionalnosti. Svaki tip shadera ima odgovarajući shader *procesor* u GPU-u koji podržava izvođenje tog tipa shadera (moguće je da i svi budu združeni u tzv. *Unified Shader* arhitekturi).

Sa slike 2 se jasno vidi da, ukoliko izdajemo apstrahiranu naredbu za iscrtavanje geometrije (*Draw*), prvi shader koji se pokreće nad njome je vertex shader čija je zadaća, kao što mu ime kaže, procesirati

vrhove (engl. *vertex*) svake pojedine primitive na koju je primijenjen. Takvo procesiranje u uobičajenom 3D renderiranju obvezno uključuje množenje MVP matrice iz [1] (formula (53)) s ulaznim 3D vektorom vrha u model-prostoru, no može uključivati (linearno ili ne) bilo kakvo dodatno procesiranje vrhova u kojem god inicijalnom prostoru se oni nalazili pri predavanju GPU-u na iscrtavanje. U vrijeme prije programabilnih shadera, ovu nužnu transformaciju (koju je nVidia tada nazvala *Transform and Lighting; TnL* jer su bili prvi koji su podržavali takvo procesiranje, uključujući i fiksne efekte sjenčanja, na akceleratoru) je obavljala neka fiskna jedinica na grafičkom akceleratoru (dakle, bez mogućnosti *sasvim* proizvoljnih transformacijskih operacija) ili pak sam CPU ukoliko akcelerator to nije podržavao (najstariji modeli). Sa strane rasterizacije je jedino bitno da rezultat primjene vertex shadera na dani vrh uvijek bude *clip space* NDC koordinata te točke s kojom se onda može lako manipulirati, što omogućuje buduće računanje konačne piksel koordinate, rasterizaciju primitiva i sl. Valja napomenuti da vertex shader uvijek obrađuje točno *jedan* vrh primitive i nikako ne može pristupati ostalima, mada sam vertex shader može biti pozvan više od jedanput za pojedini vrh. Sljedeći na slici je shader za kontrolu teselacije te njegov „brat", shader za evaluaciju teselacije. Ova dva shadera su zanimljiva po tome što zajedno omogućuju generiranje složenije geometrije od one koja je predana vertex shaderu prije; točnije, ta ulazna geometrija se tipično naziva *zakrpom* (engl. *patch*) i daje okvirnu, nisko-rezolucijsku primitivu (trokut, četverokut ili izoliniju) za daljnju teselaciju. Kontrolni shader ovdje ima ulogu programa koji se izvodi za svaki pojedini vrh svake pojedine ulazne zakrpe te odlučuje koje vrhove pustiti u izlaznu zakrpu s proizvoljnim transformacijama i računima, kao i koje vrhove preskočiti. Nakon toga, fiksna funkcionalnost generatora primitiva teselacije (engl. *Tessellation Primitive Generator*) se pobrine da se dobivene zakrpe subdiviraju na željeni način, sa željenom maksimalnom razinom teselacije i ostalim parametrima, nakon čega shader za evaluaciju teselacije prima pojedine vrhove od te nove generirane „gusto" raspoređene primitive (slično vertex shaderu) i računa nove pozicijske koordinate, teksturne koordinate i bilo koje druge podatke za koje se smatra da imaju drukčiju vrijednost u interijeru početne zakrpe, što znači da dovode nove detalje u polaznu geometriju. Na taj način ova dva shadera u tandemu omogućavaju naprednije, brzo renderiranje modela definiranih u obliku NURBS ili sličnih splajnova, parametarskih krivulja i dr., u potpunosti na GPU-u bez dodatne komunikacije s CPU-om, što također znači da je moguće progresivno teselirati određene modele ovisno o udaljenosti, njihovoj projiciranoj površini na ekranu i dr. parametrima. Uočimo da okolne strelice koje u potpunosti zaobilaze teselacijski segment na 2 znače da teselacijski shaderi nisu nužni pri renderiranju i ne moraju biti povezani i spojeni u GPU program koji se izvodi za izdanu operaciju iscrtavanja. Na sličan način vidimo da je moguće zaobići i geometrijski shader, koji također nije obvezan.

Sljedeći na redu je geometrijski shader koji ima ulogu programa koji operira nad jednom ulaznom primitivom i od nje generira i emitira nula ili više izlaznih primitiva. To omogućava renderiranje relativno složenih modela u potpunosti na GPU-u, bez potrebe da se s aplikacijske strane stalno šalju isti podaci za jedan te isti model koji može biti vrlo složen i čiji transfer s glavne memorije i CPU-a na GPU može biti dugotrajan. Umjesto da se stalno prenose isti podaci, GPU-u se može dati do znanja da treba renderirati samo npr. jednu točku (najjednostavnija i najjeftinija primitiva) s primijenjenim geometrijskim shaderom koji će od te pseudo-točke generirati trokutove za npr. utaški čajnik. Sa slike 2 također možemo uočiti da je geometrijski shader dvosmjerno povezan sa spremišnim međuspremnikom (*Storage Buffer*) preko zajedničke sabirnice koja omogućava da se njegovi izlazni podaci pohrane interno u GPU memoriji i da bez potrebe za komunikacijom s CPU-om renderira generiranu geometriju koliko god je puta potrebno ili uz koliko god prolaza (engl. *pass*) je potrebno radi nekog specijalnog efekta. Štoviše, toj memoriji preko posebnih tehnika može pristupiti i CPU, tako da geometrijski shader može poslužiti i za odvijanje općenitih računskih operacija na GPU-u, mada to više nije optimalno ukoliko su dostupni računski shaderi. Nakon rasterizacije konačno generiranih primitiva i pojedinih pred-fragmentskih operacija, slijedi nužna primjena fragment shadera (ili piksel shadera) čiji je zadatak možda i najvažniji sa stajališta vizualizacije: odrediti boju trenutnog fragmenta ili piksela na framebufferu. Naravno, taj program mora imati pristup raznim drugim resursima koje možemo zamisliti da su potrebni u ovoj ključnoj fazi: teksture, razne dodatne tablice, invarijantni (*uniform*) skalari/vektori/matrice ili drugi parametri koji se ne mijenjaju tijekom rasterizacije danog objekta, općenit pristup spremišnoj memoriji itd. – sve te veze

možemo vidjeti na slici 2. Osim invarijantnih podataka, fragment shaderu su gotovo uvijek potrebni i promjenjivi podaci (*varying*), tj. oni podaci koje je potrebno perspektivno-korektno linearno interpolirati uzduž cijele primitive te evaluirati vrijednost na pojedinoj koordinati fragmenta. Tako funkcionira mapiranje tekstura – pojedinačne (u,v) koordinate su dodijeljene vrhovima trokuta kao atributi (*attribute*) u vertex shaderu, kako bi se interpolirali preko trokuta prilikom rasterizacije i kako bi ih fragment shader mogao evaluirati na pojedinom fragmentu te iskoristiti konačnu (u,v) koordinatu trenutnog fragmenta za dohvat željenog teksela iz videomemorije (vidljivo iz *Sampled Image* i *Vertex Shader* veze s fragment shaderom). Pored toga, fragment shaderi su oni koji su najviše zaslužni za sve vizualne efekte u raznim prezentacijama, videoigrama i drugim interaktivnim 3D iskustvima – od glatkih sjena, naprednog sjenčanja i osvjetljenja, refrakcija, do raznih post-procesnih efekata poput zamućenja cijelog ekrana, isijavanja svjetlih površina, primjene kolor-filtara i sl. – sve to na kraju zahtjeva uporabu fragment shadera, ali može ovisiti i o mnogim drugim shaderima, ovisno o implementaciji i željenim performansama. Pojam *fragment* i *piksel* ovdje koristimo u podjednakom smislu, zato što nismo napravili razliku između renderiranja bez bilo kakve vrste *supersampling*-a odnosno renderiranja na veću rezoluciju koja se tada filtrira i smanji na ciljnu, od renderiranja $1:1$, direktno na željenu rezoluciju gdje je $1\text{piksel} = 1\text{fragment}$. Grafički API-jevi tipično nativno podržavaju barem jednu vrstu *supersampling*-a radi implementacije anti-aliasinga, a to je tzv. MSAA (engl. *Multi-sampled Anti-aliasing*) koji zahtjeva da se jedan ekranski piksel dijeli na određeni broj fragmenata, tako da se rasterizacija može djelomično (uz određene trikove i izbjegavanje nepotrebnih računa u koje ovdje nećemo ulaziti) izvesti nad takvim grupama fragmenata, na svakom pojedinom fragmentu te naposljetku filtrirati i smanjiti te grupe na originalnu dimenziju jednog piksela – na taj način se rješava problem nazubljenosti tj. aliasinga na renderiranim slikama, nastalog zbog prevelike prostorne frekvencije koja nastaje između velikog kontrasta interijera jednog i interijera drugog susjednog rasteriziranog trokuta ili druge primitive, koja po Shannon-Nyquistovom teoremo o uzorkovanju tada može dovesti do vidljivih lažnih viskih frekvencija (nazubljenja) na 2D signalu (konačnoj slici). MSAA pokušava to rješiti konačnim supersamplingom, što teoretski nikada neće sasvim rješiti ovaj problem, ali dovoljno visoke vrijednosti kombinirane s velikim PPI vrijednostima ekrana i renderiranjem u nativnoj rezolucijoj u praksi funkcioniraju dobro (izuzev problema s performansama i nekompatibilnosti MSAA-a s pojedinim naprednijim tehnikama renderiranja). Posljednji preostali tip shadera jest ujedno i najposebniji – računski shader, koji služi prvenstveno za obavljanje općenitih visoko-paralelnih računskih operacija na GPU-u. Takvi se računi tipično nazivaju GPGPU (danas je općenit naziv *compute* za sve apstraktne zadatke računanja koji su često i heterogeni u smislu da ih može izvoditi i GPU i CPU iz istog izvornog koda na koherentan način) zadacima te su posebno pisani i optimizirani tako da iskoriste masivnu paralelnost današnjih GPU-ova u cilju što bržeg obavljanja zadataka koji se mogu dovoljno dobro rastaviti na nezavisne podzadatke. Tu dolazi do izražaja niska memorijska latencija GPU-a koji najčešće mogu nastaviti obavljati operacije u nekoj drugoj „niti" dok trenutna nit ne dobije željeni podatak iz memorije, kao i ostale karakteristike GPU-ova (brže upravljanje nitima koje je ugrađeno u sam hardver, kontrola redoslijeda izvođenja pojedinih shadera integrirana u GPU jedinice i sl.) koje idu u prilog njihovoj primjeni na visokoparalelne zadatke koji na tipičnim višejezgrenim CPU-ovima današnjice još uvijek ne postižu zadovoljavajuće performanse. Sve to znači da su ovi shaderi jako drukčiji od svih ostalih (osim možda onih za kontrolu teselacije, koji imaju sličan opći način rada) po svojem funkcioniranju i po tome kako se kodiraju. Ovakvi računski zadaci se mogu odvijati nezavisno od zadataka iscrtavanja geometrije, što je vidljivo sa slike 2 po tome što računski zadaci počinju na sasvim zasebnom slijedu operacija, nepovezanim s onima za iscrtavanje lijevo. Dakle, moguće je obavljati intenzivne račune na GPU-u bez da se išta iscrtava u framebuffer. Očekivano, računski shader ima pristup svim međuspremnicima i pohranjenim slikama u GPU memoriji preko zajedničke sabirnice; doduše, zapisivanje je moguće samo u one memorijske zone koje služe kao općenita spremišta podataka, dok je ostale zone moguće samo čitati – u njih se tipično može zapisivati samo s aplikacijske strane, putem odgovarajućih API poziva (koji postavljaju *uniform* vrijednosti i slično). Naravno, rezultate koje računski shader zapisuje (u *Storage Buffer* ili drugdje) je moguće čitati s aplikacijske strane.

Konačno, na slici 2 vidimo plave, svjetloplave i zelene blokove koji predstavljaju razne klase objekata

pohranjenih u videomemoriji GPU-a, potrebnih za izvođenje svih shadera koji imaju potrebu pristupati pojedinim objektima u toj memoriji. Tako imamo razne međuspremnike koje programer puni s aplikacijske strane kako bi inicijalno „hranio" vertex shader vrhovima geometrije za scenu. Oni uključuju međuspremnik vrhova (*Vertex Buffer Binding* – jednostavno pohranjuje sve tipično 3D model-prostorne vektore vrhova geometrije u sceni; napomenimo da je ovo samo jedan potencijalno jako dugi niz brojeva te stoga nije moguće samo na osnovu njega utvrditi kojem modelu kojeg želimo iscrtati pripada koja koordinata vrha) te međuspremnik indeksa (*Index Buffer Binding* – koristi se kako bi se riješio iznad spomenuti problem dugog niza brojeva; ovaj međuspremnik na *n*-toj lokaciji praktički sadrži „pokazivač" na točnu lokaciju u međuspremniku vrhova na kojoj možemo pronaći *n*-ti vrh za trenutni model; taj međuspremnik je prije ispunjen s aplikacijske strane s potrebnim nizom koordinata vrhova i možda još nekim podacima atributa vrhova, zajedno s pokazivačima atributa koji su usmjereni na odgovarajuće identifikatore vršnih atributa u vertex shaderu kako bi GPU znao pristupiti točnom atributu preko predanog indeksa). Također vidimo *Sampled Image* i *Storage Image* slikovne objekte u memoriji, koji omogućuju baratanje s 2D slikama (to je koncept različit od tekstura koje pak predstavljaju skup tj. sloj nekoliko slika u smislu mipmap niza) od strane pojedinih shadera. Potonji tip slika dozvoljava i čitanje i pisanje, dok prvi može služiti samo za čitanje podataka koji se mogu koristiti kao tekseli unutar fragment shadera ili općenito kao neka druga vrsta podatka u bilo kojem drugom tipu shadera (u tom slučaju uz neke restrikcije). Naposljetku imamo *Push Constants*, relativno novi koncept koji je unio Vulkan API i koji omogućava jako efikasan prijenos konstanti shaderima od strane aplikacije [5], prvenstveno radi fleksibilnosti oko promjene stanja geometrije koje inače u nekim situacijama može nedostajati. Takve konstantne je najbolje koristiti za male, često-mijenjane vrijednosti u shaderima poput matrica za skeletalnu animaciju tijela i sl.

3 Implementacije rasterizatora

Sada ćemo se fokusirati na samo jedan korak (operaciju) sa slike 2, naime *rasterizaciju* te na različite uobičajene i moguće pristupe njenoj *softverskoj* realizaciji. Drugim riječima, opisujemo sve što se može događati unutar *Rasterization* bloka sa slike pod pretpostavkom da su sve primitive već složene i svi prethodni fiksni i programabilni stadiji uspješno obavljeni, pri čemu ignoriramo sve korake nakon rasterizacije osim dubinskog testiranja (pošto želimo korektno iscrtavanje proizvoljnih scena bez posebnog sortiranja primitiva). Dakle, sve što preostaje jest obrezivanje ulaznih primitiva, njihovo konačno transformiranje u 2D piksel koordinate ekranskog prostora te konačno ispunjavanje primitiva zadanom bojom koja može varirati na svakom vrhu ili pak mapiranje tekstura na primitivu. To je znatno pojednostavljena verzija općenitog rasterizatora koji bi trebao podržavati rad s piksel shaderom koji računa konačnu boju piksela, no ovdje se ne bavimo shaderima već samo rasterizacijom pa ovaj rasterizator možemo shvatiti kao softverski ekvivalent prvim hardverskim rasterizatorima od prije 20 i više godina, koji su možda imali još pokoju opciju poput miješanja boja pojedinih piksela ili neku vrstu anti-aliasinga. Dodavanje piksel shadera na konačnu implementaciju koja je dio ovog rada je jednostavno koliko i analiziranje, parsiranje te interpretiranje/izvođenje generiranog (među)koda od početnog izvornog jezika (kojeg god da se uzme, najvjerojatnije GLSL) te uključivanje mnogih dodatnih objekata u API poput `sampler`-a i dr., što očito nije trivijalan zadatak zbog mnogo različitih koncepata koje sadašnji API-ji poput OpenGL-a podržavaju nakon mnogo godina razvoja. Ni ne spominjemo ostale shadere jer oni bi zahtjevali još više posla koji trenutno nije fokus ovog rada.

Cilj ove sekcije je predstaviti kratku povijest rasterizatora koji su namijenjeni za rad u realnom vremenu te njihovu evoluciju tijekom vremena zbog drastičnih promjena u dostupnom hardveru. Proći ćemo kroz tri glavna pristupa izrade rasterizatora, njihove prednosti i mane u odnosu na današnja računala te ćemo implementirati zadnje obrađeni tip rasterizatora i komentirati neke njegove karakteristike u kontekstu današnjih grafičkih procesora.

3.1 Konačne transformacije vrhova

Prije nego što krenemo s pojedinim algoritamskim pristupima samoj rasterizaciji, potrebno je riješiti pitanje koordinata vrhova geometrije koja je došla do ovog koraka. Naime, pretpostavljamo da su sve operacije koje su do sada provedene nad 3D točkama geometrije bile na krajnjem koraku vertex shadera transformirane perspektivnom MPV matricom koja ih je dovela u NDC prostor, pri čemu je polazna z koordinata transformirana bilo kojom funkcijom pseudo-udaljenosti koja nam garantira svojstva iz [1] (poglavlje 3.3.3), vrlo važna radi točnog dubinskog sortiranja piksela. Sada možemo lako transformirati točku (x,y) iz $[-1,1]\times[-1,1]$ NDC kvadrata (ignoriramo z koordinatu nakon implicitno obavljene ortografske projekcije jer nam nije potrebna tijekom rasterizacije izvan uzorkovanja interpolirane z koordinate koja se zato može smatrati atributom vrhova primitiva) u odgovarajući $w\times h$ pravokutnik prozora s donjim lijevim vrhom u $(0,0)$. Očito, odgovarajuća afina transformacija u točku (x',y') je dana sa:

$$x' = \frac{x+1}{2}w \tag{1}$$

$$y' = \frac{y+1}{2}h. \tag{2}$$

Ovo međutim još uvijek ne predstavlja odgovarajuće *indekse* u 2D polju ekranskih piksela, koji moraju biti cijeli brojevi iz segmenta $[0,w-1]$ odnosno $[0,h-1]$. Pošto želimo da to polje indeksira *središta* pojedinih piksela na ekranu, a x' i y' su uvijek pozitivni, dovoljno je za indekse uzeti

$$i = \lfloor x' \rfloor, \text{ odnosno} \tag{3}$$

$$j = \lfloor y' \rfloor. \tag{4}$$

Ipak, ove jednadžbe za $x' = w$ (x' poprima tu vrijednost za $x = 1$, što je moguće) daju $i = w$, što je izvan raspona, pa u tom jedinom posebnom slučaju definiramo da mora biti $i = w-1$; slično, zbog situacije s y' definiramo $j = h-1$ kada $y' = h$. Važno je reći da kada u bilo kojem algoritmu uzimamo postojeći piksel-indeks i njime računamo neku novu piksel koordinatu, na kraju računa moramo osigurati da smo „sletili" točno na *središte* odgovarajućeg piksela, a ne na neku drugu proizvoljnu lokaciju! Stoga je potrebno, prije primjene (3) na trenutne koordinate, dodati $\frac{1}{2}$ objema koordinatama, kako bismo se npr. smjestili iz koordinate donjeg lijevog ruba piksela točno u njegovo središte. Ako nismo bili na tako ekstremnoj poziciji, dodavanje polovine će uvijek osigurati da nakon primjene (3) završimo na središtu *najbližeg* piksela, što uvijek vrijedi jer su koordinate pozitivne, a dodavanje $\frac{1}{2}$ prije primjene najveće-cijelo-od funkcije jest očito ekvivalentno zaokruživanju na najbliži cijeli broj (svi brojevi s decimalnim dijelom $\geq \frac{1}{2}$ završe na idućem cijelom broju, dok svi ostali ostanu na trenutnom čistom cjelobrojnom dijelu). Dakle, vrijedi

$$[x] = \left\lfloor x + \frac{1}{2} \right\rfloor \tag{5}$$

i tu ćemo formulu koristiti.

3.2 Bresenhamov algoritam

Jedan on najstarijih rasterizacijskih algoritama je Bresenhamov algoritam, izvorno zamišljen 1962. radi iscrtavanja segmenta pravca na 2D rasteru, a koji se može primijeniti i za rasterizaciju trokuta te linearnu interpolaciju raznih atributa njegovih vrhova. On je zamišljen u vrijeme kada računala nisu mogla obavljati brzo dijeljenje cijelih brojeva, kamoli onih s pomičnim zarezom, stoga je ovaj algoritam vrlo efikasan sa stajališta primitivnih operacija koje obavlja, budući da ne zahtijeva niti cjelobrojno dijeljenje ni množenje (u osnovnoj varijanti). Za početak, uvjerimo se da svi već sada iz osnova geometrije znamo naivno implementirati rasterizaciju segmenta (ne pazeći na to koje operacije pritom koristimo), pod pretpostavkom da su 2D indeksi početnog odnosno krajnjeg piksela (i_1,j_1) odnosno (i_2,j_2) te da vrijedi $i_1 < i_2$ i $j_1 \leq j_2$ te $\frac{j_2-j_1}{i_2-i_1} \leq 1$ (uklanjanjem ovih restrikcija se bavimo nešto kasnije):

```
1  float dy = j2-j1;
2  float dx = i2-i1;
3  float j = j1;
4  float k = dy/dx;
5
6  renderpix(i1,j1); //ova funkcija jednostavno zapisuje
7  //neku konstantnu boju linije na indeksu (i1,j1);
8  //određivanjem točne boje u složenijim situacijama
9  //se bavimo kasnije
10 for(int i = i1+1; i1 <= i2; ++i)
11 {
12   j += k;
13   renderpix(i, static_cast<int>((std::round(j)))); //primjena (5)
14 }
```

Gornji kôd funkcionira, no pritom koristi dijeljenje brojeva s pomičnim zarezom na liniji 4, kao i njihovo zbrajanje na 12. liniji. Te operacije na današnjim procesorima nisu puno sporije od cjelobrojnih, no još prije 20 i više godina obavljanje takvih operacija u aplikacijama koje intenzivno rasteriziraju blizu milijun piksela 30 ili više puta u sekundi, na mikroprocesorima (bez ikakvog paralelizma) bez (brzih) FPU-a, sigurno nije bila dobra ideja. Stoga treba pronaći način da se ukloni potreba za `float` tipom varijabli i po mogućnosti u potpunosti izbjegne dijeljenje koje je u svakom slučaju najskuplja od osnovnih aritmetičkih operacija koje CPU direktno podržava.

Takvo je rješenje upravo Bresenhamov algoritam, koji koristi gore spomenute restriktivne pretpostavke na ulazni segment kako bi u potpunosti izbjegao gornju liniju 4. Naime, određivanje koeficijenta smjera je nužno kako bismo znali koliko vertikalnih jedinica dodati u svakoj iteraciji petlje (tj. pri svakom koraku udesno na segmentu), no zbog pretpostavki znamo da je taj omjer uvijek ≤ 1, što znači da mi ustvari niti u jednoj iteraciji ne dodajemo jedan cijeli broj na indeks j, već samo neki decimalni dio. To možemo iskoristiti tako da uzmemo u obzir činjenicu da kada koeficijent smjera pomnožimo sa dx dobivamo dy, koji je očito cjelobrojan, te da tada u liniji 12 možemo jednostavno zbrajati dy. Također, primijetimo da se u gornjem algoritmu zaokruživanje na sljedeći veći cijeli broj u liniji 13 događa samo kada je j prešao $\frac{1}{2}$, koji u sadašnjem algoritmu poprima vrijednost $\frac{dx}{2}$. Dakle, kombiniranjem tih činjenica uočavamo da možemo započeti „akumuliranje" varijable od 0 i da ona neće moći odmah biti veća od $\frac{1}{2}$ zato što je $\frac{dy}{dx} \leq 1$ (događaj prelaska preko $\frac{1}{2}$ se može dogoditi najviše samo jedanput po iteraciji). Kada pređe $\frac{1}{2}$, to znači da je vrijeme da j skoči na sljedeći cijeli broj, a akumulator se tada mora vratiti za 1 piksel unatrag jer je upravo za toliko sada napravljena korekcija konačnog j indeksa; ovdje 1 piksel znači dx jedinica pošto smo iz prošlog algoritma sve praktički množili s dx. Preostaje jedino dokazati da računanje praga $\frac{dx}{2}$ *cjelobrojnim* dijeljenjem ne čini ovaj algoritam krivim; provjerimo dva moguća slučaja: kada je dx paran, odnosno neparan. U prvom slučaju očito nema razlike od općenitog dijeljenja; u drugom, dobivamo za rezultat cijeli broj za $\frac{1}{2}$ manji od točnog racionalnog broja, što implicira da u provjeri da li je akumulator prešao $\frac{dx}{2}$ moramo dobiti podjednak logički rezultat, jer se akumulator povećava za $dy \geq 1$, pa se ne može dogoditi da je veći od $\frac{dx}{2} - \frac{1}{2}$ a da nije veći od $\frac{dx}{2}$. Ovime je dokazana ekvivalencija Bresenhamovog algoritma i algoritma 1. Slijedi implementacija u C++-u:

```
1  int dy = j2-j1;
2  int dx = i2-i1;
3  int akumulator = 0;
4  int prag = dx/2;
5  int j = j1;
6
7  renderpix(i1,j1);
8  for(int i = i1+1; i1 <= i2; ++i)
9  {
10   akumulator += dy;
11   if(akumulator > prag)
12   {
13     ++j;
14     akumulator -= dx;
15   }
16   renderpix(i,j);
```

```
17 }
```

U liniji 4 se cjelobrojno dijeljenje može zamijeniti jednim bit-pomakom udesno, čime se u potpunosti izbjegava dijeljenje i čini ovaj algoritam vrlo efikasnim (ovdje očekujemo da to automatski napravi optimizirajući kompajler). Mada je ovaj algoritam ovdje prikazan kao algoritam za iscrtavanje segmenata, on je koristan i kao osnovni blok koda koji se može koristiti za linearnu interpolaciju u rasterizatorima. Sljedeći kôd prikazuje takvu primjenu Bresenhamovog algoritma; može se koristiti za interpolaciju proizvoljnih atributa vrhova trokuta ili segmenata (primitiva) uzduž cijele površine/dužine, pod uvjetom da je taj atribut linearan u trenutnom 2D ekranskom prostoru. Ovdje malo mijenjamo polazne pretpostavke: pošto općenito nije moguće zahtijevati da atributi zadovoljavaju bilo kakve uređajne relacije na vrhovima primitiva (kao što smo to mogli napraviti za njihove x i y koordinate), ovdje ne možemo pretpostaviti da je $z_1 \leq z_2$. Atributi ovdje moraju biti striktno cjelobrojni, te je zbog ovih novih generalizacija potrebno obaviti cjelobrojno dijeljenje na matematički korektan način tj. tako da se rezultat dijeljenja uvijek (formalno) ispravno provede kroz najveće-cijelo-od funkciju; C++ aritmetički operatori ne specificiraju takvo ponašanje, pa moramo prvo napisati jednu pomoćnu funkciju. I dalje pretpostavljamo da je $x_1 \leq x_2$.

Listing 3: Matematički ispravna implementacija cjelobrojnog dijeljenja

```
1  template <class T>
2  T idiv(T a, T b) //pretpostavljamo da b!=0
3  {
4    static_assert(std::is_integral<T>::value, "Ovu␣funkciju␣smijete␣pozvati␣samo␣s␣
         cjelobrojnim␣argumentima!");
5    if(a > 0 && b < 0 || a < 0 && b > 0)
6      return a/b-1;
8
8    else
9      return a/b; //C++11 je specificirao ponašanje cjelobrojnog dijeljenja da uvijek
           odbacuje decimalni dio rezultata dijeljenja
10 }
```

Listing 4: Općenit Bresenhamov algoritam za linearnu interpolaciju atributa na segmentu

```
1  int dz = z2-z1;
2  int dx = i2-i1;
3  int akumulator = 0;
4  int prag = dx/2;
5  int q = idiv(dz,dx);
6  int mod = dz - q*dx;
7  int z = z1;
9
9  eval(i1,z1); //ova funkcija smješta izračunatu vrijednost atributa na i1 x-koordinati u
       neki međuspremnik
10 for(int i = i1+1; i1 <= i2; ++i)
11 {
12   z += q;
13   akumulator += mod;
14   if(akumulator > prag)
15   {
16     ++z;
17     akumulator -= dx;
18   }
20
20   eval(i,z);
21 }
```

U liniji 6 vidimo da računamo ostatak dijeljenja koje je sada neophodno budući da atributi mogu imati praktički proizvoljne koeficijente smjera. To znači da je jedina promjena u odnosu na 2 ta što osim akumulatora imamo i kvocijent `q` koji jednostavno dodajemo interpolandu u svakoj iteraciji u liniji 12. Pored toga, ostatak `mod` ima istu ulogu kao `dy` u 2 jer zadovoljava uvjet da je $\frac{mod}{dx} \leq 1$, što slijedi iz definicije cjelobrojnog dijeljenja. Dakle, nakon dodavanja kvocijenta moramo jedino nastaviti dodavati `mod` do

praga, analogno algoritmu 2. Primijetimo da zbog načina funkcioniranja Bresenhamovog algoritma, ovaj algoritam zaokružuje vrijednosti interpoliranih atributa na najbliži cijeli broj. Spomenuta `eval` funkcija je ovdje samo simboličke prirode, kako bismo mogli prikazati ovaj algoritam u izolaciji; u stvarnosti bismo morali ovaj kôd prožeti kroz standardnu x, y rasterizaciju iz 2 tako da se z interpolacija zbiva u tandemu s onom od y koordinate, što je očito moguće napraviti unutar iste `for` petlje, s dva različita akumulatora.

Objasnimo sada kako se riješiti prethodno spomenute tri restriktivne pretpostavke na Bresenhamov algoritam 2. Općenito, to je moguće napraviti sljedećim nizom transformacija koji može služiti kao svojevrsni algoritam za postavljanje primitiva u stanje u kojemu se mogu rasterizirati uporabom Bresenhamovog algoritma.

1. Ako je $i_1 > i_2$, zamijeniti vrhove trenutnog segmenta;
2. Ako je $j_1 > j_2$, uzeti `dy=j1-j2;` u liniji 1, zamijeniti sadržaj linije 13 s `j--;`
3. Ako je $\Delta j > \Delta i$, zamijeniti uloge `i1` i `j1` te `j1` i `j2`.

Teorijski, implementacija 2 ne smanjuje općenitost dok god nekako možemo prije rasterizacije pojedinog segmenta obaviti sve tri gornje provjere za redom. Prva transformacija se može implementirati sa jednom usporedbom, dakle jednim grananjem i dvama zamjenama (`i1` s `i2` te `j1` s `j2`). Druga transformacija u biti zahtjeva dvije ekvivalentne operacije – množenje s -1 – prvo sa `dy` a zatim s imaginarnom jediničnom varijablom koja služi za operaciju inkrementiranja u liniji 13. Iz tog razloga, 2 bi trebalo implementirati na drukčiji način (ako se želi iskoristiti baš ova optimizacija) tako da linija 13 sadrži zbrajanje `j` s još jednom varijablom koja je ovoj proceduri prenesena od strane postavljača primitiva; ta varijabla ima vrijednosti -1 ili 1, već prema tome treba li obaviti ovu transformaciju ili ne. Usto, ista bi mistična varijabla morala množiti `dy` u liniji 1, što znači da ova transformacija „košta" jedno (uvjetno, jasno) grananje (množenje nećemo računati u cijenu jer je ono uvijek prisutno u kodu, samo što nema efekta kada je varijabla vrijednosti 1 tj. kada transformacija nije potrebna). Posljednja transformacija u nizu provjera (potrebna za renderiranje npr. vertikalnih segmenata) zahtjeva unakrsnu permutaciju indeks-varijabli, koja se opet može implementirati pomoću dvije zamjene: `i1` s `j1` i `i2` s `j2`. Vidimo da sve transformacije uzimaju po jedno grananje, s time da prva i treća obavljaju još po dvije zamjene svaka. To znači da će za svaki podneseni segment biti potrebno napraviti tri grananja samo za postavljanje segmenta za rasterizaciju, bez da uopće gledamo sam Bresenhamov algoritam. To je dosta dobar rezultat, no postoje drukčiji pristupi rasterizaciji koji uopće ne zahtijevaju ovu vrstu transformacije segmenata i stoga ne dovode procesor do nepotrebnih promašaja cache-a zbog krivih predviđanja grananja, što može dosta negativno utjecati na performanse kada se renderiraju scene s jako puno primitiva, najčešće trokutova. U takvim situacijama puno trokutova može imati jako različite orijentacije segmenata (tj. projiciranih stranica trokuta) koje su praktički nasumično raspoređene na ekranu te mogu dovesti do puno promašaja cachea. Povrh toga, u 4 vidimo da primjena Bresenhamovog algoritma za interpolaciju općenitih atributa zahtijeva da oni budu cjelobrojni kako bi se mogle iskoristiti glavne prednosti ovog algoritma, no pošto praktički niti jedan atribut u najčešćim primjenama nije cjelobrojan, sve prednosti padaju u vodu jer bi se ionako trebalo računati s `float` brojevima ili brojevima s fiksnim zarezom za obavljanje interpolacije, pa se moglo i odabrati neki drugi algoritam umjesto Bresenhamovog čiji je glavni adut izbjegavanje množenja i dijeljenja sada anuliran. Procesori su s vremenom postajali sve brži u pogledu cjelobrojnog dijeljenja i množenja, pa je cijena dodatnog grananja i nedostatka općih atributa vrlo brzo nadišla onu koju nude drugi algoritmi koji koriste dijeljenje. Ne pomaže ni to što nakon obavljenih transformacija, linija 11 u svakoj iteraciji petlje u 2 obavlja još jedno grananje, što znači da smo zamijenili cjelobrojno dijeljenje s po mogućnosti *još više* grananja, koja na novim arhitekturama mogu dovesti do ozbiljnih „mjehurića" u dubokim CPU cjevovodima. Sve su to razlozi zašto već i prije više od 20 godina ovaj algoritam nije bio omiljen među implementatorima softverskih rasterizatora, poglavito u industriji videoigara koja je tada doživljavala veliku grafičku revoluciju i zahtjevala jako brze implementacije na široko dostupnim procesorima koji su tada počeli dobivati integrirane FPU-ove, što je još jedna motivacija za napuštanje Bresenhamovog algoritma. To nipošto ne znači da taj algoritam nije

koristan; on je na neki način temelj izgradnje bilo kojeg dijela rasterizatora u sljedećoj sekciji i osnovni 2 kôd može i dalje poslužiti npr. za šetnju po stranicama trokuta, mada nije optimalan za interpolaciju drugih vrijednosti osim piksel koordinata. On je i dalje najjednostavnija i najefikasnija opcija za iscrtavanje linijskih primitiva (ne trokuta) koje nisu anti-aliasirane (uz mogućnost konstantne i interpolirane boje), ali ako želimo glatke, anti-aliasane linije proizvoljne debljine, postoje bolji algoritmi koji vrlo efikasno obavljaju filtriranje segmenata poput Gupta-Sproull algoritma i dr. Ovdje implementiran Bresenhamov algoritam operira nad pojedinačnim segmentima; za trokut-primitive, potrebno je jedino sortirati vrhove prema ekranskoj y-koordinati i generirati horizontalne segmente uzduž dvije suprotne stranice trokuta koje se zatim interpoliraju i iscrtaju. Sve te korake je moguće obaviti istim algoritmom 2 (ako se koriste samo cjelobrojni atributi), no sada prelazimo na jedan malo drukčiji tip rasterizatora koji omogućava efikasnije iskorištavanje nešto novijeg hardvera (ne s današnjeg stajališta!) za iscrtavanje trokutova s atributima koji mogu biti proizvoljni brojevi s pomičnim zarezom.

3.3 Scanline rasterizator s fiksnim zarezom

Sredinom 1990-ih su PC-jevi postali dovoljno moćni za izvođenje renderiranja potpuno trodimenzionalnih scena u potpunosti na CPU-u (softversko renderiranje). Tome je doprinio izrazito brz razvoj mikroprocesora u to doba, predvođen Intelovom Pentium mikroarhitekturom koja je donijela dva nezavisna cjevovoda zajedno sa sasvim integriranim FPU-om (superskalarni procesor) na jedan CPU čip. Promjene koje je to omogućilo u softveru uključuju interaktivnu (za ondašnje vrijeme; možda 20 fps) 3D rasterizaciju nekoliko tisuća trokutova na ondašnje tipične rezolucije, otprilike 320×240. Zbog viših dostupnih taktova novih procesora, ali i relativno skupih grešaka promašaja predviđanja grananja, programeri su se počeli odmicati od Bresenhamovog algoritma i počeli koristiti sada mnogo brže instrukcije dijeljenja, kao i posebne FPU operacije, umjesto dodatnih grananja. FPU je ovdje ključan zato što omogućava transformaciju 3D geometrije u svim prethodnim stadijima tipičnog grafičkog cjevovoda, koji su se tada uglavnom svodili samo na MVP transformaciju koja se mogla puno efikasnije i preciznije izvesti koristeći aritmetiku s pomičnim zarezom, poglavito zato što su se FPU instrukcije mogle izvoditi nezavisno od standardnih ALU operacija. Sve to je zahtjevalo intenzivne asemblerske optimizacije od kojih su neke spomenute u [9]. Dva glavna pristupa rasterizaciji, koja su već spominjana u ovom radu, jesu *perspektivno-korektno* i *afino* interpoliranje atributa vrhova trokuta (i segmenata, ali više se nećemo posebno baviti segmentima pošto su oni vrlo jednostavan slučaj riješen Bresenhamovim algoritmom). Afina interpolacija atributa podrazumijeva ignoriranje činjenice da se između vrhova u svjetskom prostoru i ekranskom prostoru zbiva perspektivna transformacija koja je s gledišta standardnih točaka u euklidskom prostoru *nelinearna* transformacija; afina interpolacija svejedno i dalje izvodi interpolaciju atributa kao da su oni izravno linearni i u ekranskom prostoru, što je općenito pogrešno i dovodi do nelinearnih distorzija „savijanja" tekstura, boja ili bilo kojih drugih vizualnih atributa trokuta prilikom pomicanja kamere. Ovakva je interpolacija bila najviše korištena u ranim 3D videoigrama i sličnim aplikacijama pošto je najlakše mogla ponuditi visoke performanse. Ipak, ubrzo se pokazalo da njeni artefakti nisu baš podnošljivi za korisnike pojedinih vrsta 3D aplikacija, pa se počelo implementirati što efikasnije perspektivno-korektne rasterizatore koji interpolaciji pristupaju na sasvim korektan, ili kvazi-korektan način koji je golim okom gotovo nemoguće razlučiti od potonjeg, a koji i dalje omogućuje visoke performanse u stvarnom softveru.

Mi se ovdje općenito bavimo isključivo perspektivno-korektnim rasteriziranjem, mada u 4.3 spominjemo zanimljive primjene i posebne slučaje afine interpolacije u rasterizatorima. Prije nego što opišemo glavni algoritam, moramo navesti točne zahtjeve koje imamo od rasterizatora trokuta i uvesti neke nove aritmetičke algoritme koji se ovdje intenzivno koriste.

Od rasterizatora zahtijevamo sljedeća svojstva:

- Mora postojati bijekcija između piksela konačne slike i interijera trokuta;
- Posebno, gornje svojstvo znači da ne smije biti nikakvih rupa između susjednih trokutova;

- Smijemo pretpostaviti da točno jedan brid separira svaki trokut (stranu/lice) od jednog njegovog susjednog trokuta; svaki trokut dakle može imati najviše tri susjedna trokuta;
- Ne postoje nikakve posebne restrikcije na topologiju trokutaste mreže, osim da svi trokuti moraju biti konzistentno orijentirani u 3D *desnom* koordinatnom sustavu (tada se kaže da normala trokuta ide u smjeru interijera objekta kojeg je on dio).

Neka od gornjih svojstava su proizvoljno odabrana kako bi bilo jednostavnije opisati i implementirati rasterizator; također, prvo svojstvo vrijedi samo u našem slučaju gdje smo pretpostavili da ne obavljamo nikakve operacije miješanja boja, što znači da ne podržavamo prozirnost pa smijemo govoriti o bijekciji na ovaj način. Inače, ovo bi (vrlo važno radi korektnosti) pravilo trebalo drukčije formulirati. Važno je reći da ovaj tip rasterizatora očekuje da su sve koordinate s kojima barata već u ekranskom 2D prostoru; dakle, (1) je primijenjena na ulazne koordinate te je obavljeno perspektivno dijeljenje, ali ne i *floor* funkcija (3), tako da na kraju imamo samo (x,y) koordinate pojedinih vrhova. Dakle, nema rada s homogenim koordinatama u ovom rasterizatoru te se z koordinata smatra jednim *atributom* vrha.

3.3.1 Osnove aritmetike s fiksnim zarezom

Aritmetika s fiksnim zarezom (engl. *fixed-point arithmetic*) je jednostavnija varijanta reprezentacije realnih brojeva od brojeva s pomičnim zarezom. Radi se jednostavno o cijelim brojevima, tipično veličine koja se poklapa s podržanom (pod)veličinom strojne riječi, koji imaju implicitno postavljen zarez na predodređenoj i nepromjenjivoj bit-poziciji k. To znači da je stvarna vrijednost racionalnog broja pohranjenog u takvoj varijabli jednaka vrijednosti pohranjenog cijelog broja kroz 2^k. Prednost korištenja ovakvih brojeva jest u tome što nije potrebno koristiti FPU operacije za njihovo manipuliranje, kao i to što je aritmetika s njima komutativna i asocijativna jednako kao i stvarna aritmetika. To znači da sve operacije imaju unaprijed poznate, fiksne greške i da se uvijek možemo pouzdano vratiti unatrag od izračunatog rezultata u prošlost kroz obrnut niz aritmetičkih operacija zbrajanja i oduzimanja; nešto što nije općenito moguće raditi s pomičnim zarezom. Dakako, greške kod ovakve aritmetike mogu biti relativno vrlo velike (ovisno o k), ali su barem invarijantne za razliku od pomičnog zareza gdje variraju ovisno o eksponentu. Brojevi s fiksnim zarezom su idealni za primjene gdje nam je unaprijed poznat dinamički raspon svih vrijednosti s kojima baratamo te kada prema tome i prema maksimalnoj toleranciji na apsolutne greške možemo odabrati optimalan k.

Takvi su problemi upravo oni koje nalazimo u rasterizaciji s proizvoljnim atributima. Mada bi se moglo dogoditi da neki atribut ima ogromne vrijednosti izrazive s pomičnim zarezom ali neizrazive s fiksnim zarezom s razumnim k (ili simetrično, vrlo male vrijednosti), rasterizatori ovog tipa tipično zanemaruju tu mogućnost te direktno pretvore atribute na vrhovima s njihovih originalnih `float`/`int` vrijednosti u odgovarajuće najbliže reprezentacije s fiksnim zarezom i s tim tada interpoliraju. Ovaj moguć problem će biti riješen u idućem tipu rasterizatora. Usto, ovaj tip rasterizatora koristi brojeve s fiksnim zarezom za računanje koeficijenata smjera za šetnju po bridovima trokuta radi generiranja scan linija (horizontalnih segmenata u interijeru) te potom za linearno interpoliranje proizvoljnih atributa između krajeva scan linija radi ispunjenja trokuta (primijetite razliku od Bresenhamovog algoritma). Svi ti koraci se mogu obaviti u potpunosti koristeći ALU operacije nad brojevima s fiksnim zarezom, no optimizirane implementacije ovog algoritma tipično na lukav način u asembleru koriste mnogobrojne trikove koji iskorištavaju određene FPU operacije za pojedine pod-korake računa dok su ALU jedinice zauzete svojim odvojenim, nezavisnim računom. Takve su optimizacije postale moguće u to doba pojave zasebnih cjevovoda i superskalarnih mikroprocesora.

Mi ćemo obraditi aritmetičke operacije nad najčešće korištenim tipom brojeva s fiksnim zarezom u ovom tipu rasterizatora – 16.16. Ta notacija označava da se koristi 16 viših bitova za cjelobrojni dio te preostalih 16 bitova (pod)riječi za decimalni dio – drugim riječima, $k = 16$.

Listing 5: Implementacija klase za rad sa 16.16 aritmetikom s fiksnim zarezom

```
1  struct Fixed16 //16 bitova decimalne preciznosti
2  {
3    int32_t broj;
5
5    Fixed16() = default;
6    constexpr explicit Fixed16(int32_t vrijednost) : broj(vrijednost) {}
7    constexpr Fixed16(int16_t cjelobrojni) {SastaviOdDijelova(cjelobrojni);}
8    Fixed16(float vrijednost) //pretvorba iz float bi mogla biti brza maskiranjem mantise i
          eksponenta iz float varijable za što treba interpretirati float kao int, sto nije
         legalno moguce napraviti na najbrzi nacin u C++-u; C++20 uvodi bit_cast() koja mo
         žda u pozadini radi upravo na taj nacin...
9    {
10     float cjelobrojni;
11     float decimalni = std::modf(vrijednost, &cjelobrojni);
12     decimalni = std::ldexp(decimalni, 16); //(ldexp može biti sporiji od jednostavnog mno
           ženja s 65536.0f, moglo bi se zamijeniti...)nećemo provoditi zaokruživanje (std::
           lrint) ovdje, mada bi to dovelo do najmanje apsolutne greške pretvorbe
13     if(!std::signbit(cjelobrojni))
14     {
15       int32_t cijeli = static_cast<int32_t>(cjelobrojni) << 16;
16       broj = cijeli | static_cast<int16_t>(decimalni);
17     }
18     else
19     {
20       int32_t cijeli = static_cast<int32_t>(-cjelobrojni) << 16;
21       broj = -(cijeli | static_cast<int16_t>(-decimalni));
22     }
23   }
25
25   constexpr Fixed16(int16_t cjelobrojni,uint16_t decimalni)
26   {
27     SastaviOdDijelova(cjelobrojni,decimalni);
28   }
30
30   explicit operator float() //ovo se koristi samo za ispis i nisu važne performanse
31   {
32     float rez = Trunc();
33     float dec;
34     constexpr float tmp = 1.0f / 65536.0f;
35     if(broj >= 0)
36       dec = (broj & 0xffff);
37     else
38       dec = -((-broj) & 0xffff);
40
40     rez += dec * tmp;
41     return rez;
42   }
44
44   constexpr void SastaviOdDijelova(int16_t cjelobrojni,uint16_t decimalni)
45   {
46     if(cjelobrojni >= 0) //grananje potrebno zbog dvojnog komplementa koji nam ne dopušta
           da bez negacije jednostavno napravimo ubacivanje decimalnog dijela
47     {
48       broj = static_cast<int32_t>(cjelobrojni) << 16;
49       broj |= decimalni;
50     }
52
52     else
53     {
54       broj = static_cast<int32_t>(-cjelobrojni) << 16;
55       broj |= decimalni;
56       broj = -broj;
57     }
58   }
60
60   constexpr void SastaviOdDijelova(int16_t cjelobrojni)
61   {
62       broj = static_cast<int32_t>(cjelobrojni) << 16;
63   }
65
65   inline int16_t Trunc_Spec() //poseban slučaj odbacivanja decimalnog dijela:
         pretpostavljamo da je broj pozitivan
66   {
```

```
67      return static_cast<int16_t>(broj >> 16);
68    }
70
70    int16_t Trunc() //odbacivanje decimalnog dijela
71    {
72      if(broj >= 0)
73        return Trunc_spec();
74      else
75        return -(static_cast<int16_t>((-broj) >> 16));
76    }
78
78    int16_t Floor() //najveće-cijelo-od
79    {
80      if(broj >= 0)
81        return Trunc_spec();
83
83      else
84      {
85        if(static_cast<int16_t>(broj) == 0) //decimalni dio
86          return -(static_cast<int16_t>((-broj) >> 16));
87        else
88          return -(static_cast<int16_t>((-broj) >> 16))-1;
89      }
90    }
92
92    int16_t Round()
93    {
94      int32_t pola;
95      if(broj < 0)
96        pola = -0x8000;
97      else
98        pola = 0x8000;
100
100     Fixed16 tmp = *this;
101     tmp.broj += pola;
102     return tmp.Floor();
103   }
105
105   constexpr Fixed16 operator+=(Fixed16 b)
106   {
107     broj += b.broj;
108     return *this;
109   }
111
111   constexpr Fixed16 operator-=(Fixed16 b)
112   {
113     broj -= b.broj;
114     return *this;
115   }
117
117   constexpr friend Fixed16 operator-(Fixed16 a)
118   {
119     a.broj = -a.broj;
120   }
122
122   constexpr Fixed16 operator*=(Fixed16 b)
123   {
124     int64_t tmp = broj * b.broj;
125     tmp = tmp >> 16; // C++20 će garantirati ispravno ponašanje ovog izraza koje nam je
          tu potrebno
126     broj = static_cast<int32_t>(tmp);
127     return *this;
128   }
130
130   constexpr Fixed16 operator/=(Fixed16 b)
131   {
132     int64_t tmp = broj;
133     tmp = tmp << 32;
134     //tmp = idiv(tmp,b.broj);
135     tmp /= b.broj;
136     tmp = tmp >> 16;
137     broj = static_cast<int32_t>(tmp);
138     return *this;
```

```
139     }
141
141     constexpr Fixed16 Reciproc()
142     {
143       int64_t rez = 0x1000000000000; //tu smo mogli shiftati samo za 15 bitova i ostati
              unutar uint32_t (najvjerojatnije i dalje dovoljno precizno)
144       rez /= broj;
145       rez = rez >> 16;
146       return Fixed16(static_cast<int32_t>(rez));
147     }
149
149     inline Fixed16& operator++() //prefiksni
150     {
151       broj += 0x10000;
152       return *this;
153     }
155
155     inline Fixed16 operator++(int) //postfiksni
156     {
157       int32_t kopija = broj;
158       operator++();
159       return Fixed16(kopija);
160     }
162
162     inline Fixed16& operator--()
163     {
164       broj -= 0x10000;
165       return *this;
166     }
168
168     inline Fixed16 operator--(int)
169     {
170       int32_t kopija = broj;
171       operator--();
172       return Fixed16(kopija);
173     }
175
175     friend constexpr Fixed16 operator+(Fixed16 a, Fixed16 b)
176     {
177       a += b;
178       return a;
179     }
181
181     friend constexpr Fixed16 operator-(Fixed16, Fixed16 b)
182     {
183       a -= b;
184       return a;
185     }
187
187     friend constexpr Fixed16 operator*(Fixed16 a, Fixed16 b)
188     {
189       a *= b;
190       return a;
191     }
193
193     friend constexpr Fixed16 operator/(Fixed16 a, Fixed16 b)
194     {
195       a /= b;
196       return a;
197     }
199
199     constexpr bool operator<(Fixed16 b)
200     {
201       return broj < b.broj;
202     }
204
204     constexpr bool operator>(Fixed16 b)
205     {
206       return broj > b.broj;
207     }
209
209     constexpr bool operator<=(Fixed16 b)
210     {
```

```
211       return broj <= b.broj;
212     }
214
214     constexpr bool operator>=(Fixed16 b)
215     {
216       return broj >= b.broj;
217     }
219
219     constexpr bool operator==(Fixed16 b)
220     {
221       return broj == b.broj;
222     }
224
224     constexpr bool operator!=(Fixed16 b)
225     {
226       return broj != b.broj;
227     }
228 };
```

3.3.2 Obrezivanje i uklanjanje trokutova

Ovaj tip rasterizatora zahtijeva eksplicitno obrezivanje primitiva, što znači da je potrebno sve segmente i trokutove koji vire van ekrana odrezati na sjecištu s rubovima ekrana te nastaviti rad samo s onim dijelom primitive koji je unutar ekrana (ostatak se odbacuje). Moguće je i da su neke primitive u potpunosti van ekrana (nevidljive su), pa ih možemo jednostavno preskočiti u koraku postavljanja i tako ubrzati cijeli proces. Pretpostavljamo da je sustav (prvenstveno sama aplikacija) već do ovog stadija uklonio što je više geometrije moguće koristeći razne tehnike, tako da rasterizator ima što manje posla oko pretprocesiranja ulazne geometrije, te da su sve primitive koje su u potpunosti iza bliske ravnine obrezivanja ($z - near$) ili ispred najdalje ravnine obrezivanja ($z - far$) već uklonjene (postoji mala vjerojatnost da još neke primitive djelomično siječu neke od te dvije ravnine te ih se stoga ne može procesirati u potpunosti u 2D prostoru, no to sada zanemarujemo; vidjet ćemo jedno općenito rješenje u posljednem tipu rasterizatora). Također, sve primitive koje su suprotno orijentirane (dakle, u lijevom koordinatnom sustavu tj. čija je stražnja strana okrenuta prema kameri) su već trebale biti eliminirane u prethodnim stadijima, kao i degenerirani trokutovi. U tom slučaju, obrezivanje se može dosta efikasno implementirati i u clip space koordinatama i u konačnim 2D koordinatama; mi ćemo odabrati potonju opciju (bez pretjerane brige oko performansi) kako bismo prošli kroz jedan klasičan, povijesno važan algoritam obrezivanja – Sutherland-Hodgman.

Algoritam Sutherland-Hodgman [6] je vrlo često korišten algoritam obrezivanja u mnogobrojnim 3D i 2D aplikacijama, prvenstveno zbog svoje jednostavnosti i proširivosti na bilo koji konveksan obrezivački poligon (engl. *clip polygon*), a ne nužno samo na pravokutnik kao u našem slučaju. Isto tako, poligon koji se obrezuje ne mora nužno biti trokut, pa čak ni konveksan. Ukratko, ovaj algoritam tretira svaki brid oberzivačkog poligona redom kao pravac za koji se testira koje bridove trenutnog poligona obrezuje; na početku, svi bridovi ciljnog poligona ulaze u obzir, a nakon testiranja na kojoj strani (interijeru ili eksterijeru) se nalazi sjecište, stavlja se u listu za sljedeći korak samo ukoliko je *unutar* obrezujućeg poligona. Algoritam nastavlja iterirati za preostale bridove obrezujućeg poligona, uvijek na početku uzimajući listu točaka koja je izlaz iz prethodne iteracije te brišući tu listu, sve dok se ona može više ispuniti tijekom iteracije i dok ima još bridova obrezivačkog poligona za testirati. Algoritam koristi presjecanje dvaju pravaca u ravnini i testiranje sjecišta za pripadnost interijeru ili eksterijeru, što se može lako implementirati pomoću skalarnog produkta sa svim normalama bridova koji mora imati isti predznak, tipično pozitivan (ako se pripadanje interijeru tako definiralo; moglo je i suprotno). Dakle, posljednja iteracija koja se izvodi nad određenom listom vrhova će odrediti koji su konačno vrhovi na rubovima svih bridova obrezujućeg poligona; svi ostali dijelovi ciljnog poligona su izostavljeni na ovaj način.

Ovaj algoritam ima posebno jednostavan izgled za naš slučaj pravokutnog obrezujućeg poligona i is-

ključivo trokuta kao ciljnih poligona, samo što treba pripaziti na to da kada računamo sjecište trokuta i brida ekrana te vidimo da je ono zaista na rubu, moramo isrpavno interpolirati *sve atribute* vrhova tog trokuta i evaluirati ih na toj konkretnoj točki; to treba ponoviti za sva moguća sjecišta te na kraju od dobivenih pod-vrhova za dan trokut generirati nove trokutove. Naime, prilagodba ovog algoritma mora iterirati kroz sve bridove ekrana suprotno smjeru kazaljke na satu (isto kao što su orijentirani i trokutovi) te kako pronalazi sjecišta s bridovima, ubacuje ih na izlaznu listu upravo u tom poretku; osim sjecišta, novi vrh može postati i neki od rubova ekrana – onaj vrh koji testira pozitivno u sve tri provjere predznaka skalarnog produkta s normalama svakog brida trenutnog trokuta. Pri prelasku s jednog brida ekrana na sljedeći, možemo preskočiti zajednički vrh (rub), pošto je on već izračunat prije (sa svim potrebnim testovima i ubacivanjima točaka u listu novih vrhova po potrebi) i ne moramo ponavljati posao. Ponavljanje ovog za sve bridove trokuta nužno daje listu svih novih vrhova u ispravnoj orijentaciji, samo što oni još nisu triangulirani unutar ekrana – to se može jednostavno napraviti za opće konveksne poligone (a to je upravo trenutni slučaj; presjek konveksnih skupova je konveksan skup): generirajmo prvo trokute s trojkama indeksa vrhova iz liste $((1,2,3),(3,4,5),(5,6,7),\ldots)$, gdje je zadnji element tog podniza $(n-1,n,1)$ ili $(n-2,n-1,n)$, već prema tome je li n paran odnosno neparan. Nastavljamo graditi taj niz trokutova tako da ponavljamo isti postupak, samo ovoga puta preskačemo svaki drugi vrh iz prethodnog niza, dakle imamo $((1,3,5),(5,7,9),\ldots)$ i tako dalje nastavljamo sve dok je lista veća od tri vrha. Ovako ćemo sigurno dobiti triangulaciju polaznog konveksnog poligona i sada s njome (i ostalima od svih ostalih trokutova koji su dio scene i prošli su kroz svoje vlastito obrezivanje!) možemo preći na samo iscrtavanje trokutova bez straha da ćemo izletiti izvan memorijskog prostora framebuffera. Spomenimo i to da je testiranje predznaka skalarnog produkta moguće najbrže obaviti tako da se vektor željene točke $(x,y,1)$ množi s normalom danog brida (a,b,c) (redom koeficijenti iz jednadžbe hiperravnine), što se može smatrati 2D analogonom vektorskog produkta (engl. *2D cross product*[13]). Naime, to je ekvivalentno uzimanju skalarnog produkta $(x,y,1)$ s vektorom dobivenim na temelju koordinata krajeva brida: $(x_1,y_1,1)$ i $(x_2,y_2,1)$ preko rotacije za $\frac{\pi}{2}$, koja tada daje $(-y_1,x_1,1)$ odnosno $(-y_2,x_2,1)$. Za normalu nam još nedostaje $c=(y_2-y_1)x_1+(x_2-x_1)y1$, nakon čega uzmemo $a=y_1-y_2, b=x_2-x_1$ i na kraju izračunamo standardni skalarni produkt $(x,y,1)$ s (a,b,c) (ova operacija zadovoljava sva osnovna svojstva vektorskog produkta, osim toga što za rezultat ne daje vektor nego skalar koji je jednak 0 ako su vektori kolinearni).

3.3.3 Glavni algoritam

Sada smo došli do opisa glavnog algoritma koji izvodi konačno vidljiv rezultat – iscrtavanje trokutova. Prije nego što iznesemo algoritam, vratimo se na specificirano svojstvo 3.3. Ono je najvažnije za dobivanje korektnog rasterizatora jer ako postoji i najmanja mogućnost nastanka rupa između susjednih trokutova ili pak preklapanja piksela dvaju susjednih trokutova čiji su bridovi inače savršeno točno transformirani i odvojeni (tj. ta dva brida dijele identične jednadžbe normale do na predznak), tada imamo problem koji može uništiti korisničko iskustvo ili čak cijelu iskoristivost softvera jer se može dogoditi da se npr. pokušalo čitati određeni teksel s koordinatom koja je prešla alociranu memoriju za tu teksturu. Upravo su ti i ostali problemi oko izgradnje rasterizatora trokuta bili popularna tema u polju grafike u realnom vremenu sredinom 1990-ih, poglavito u industriji videoigara koje su jako osjetljive na takve artefakte ako dozvoljavaju potpuno slobodno kretanje u detaljnom 3D prostoru s mapiranjem tekstura na modelima. Niz članaka koji je popularizirao ovu problematiku među programerima i dao smjernice, algoritme i potrebne diskusije vezane za perspektivno-korektnu rasterizaciju trokutova u realnom vremenu (poglavito s ciljem mapiranja tekstura na trokutove) su bili djelo Chrisa Heckera[5] u Game Developer Magazine-u [7] te će nam kôd u njima poslužiti za osnovu ovog tipa rasterizatora.

Očito je da problemi sa 3.3 mogu nastati samo na rubovima trokuta, odnosno na bridovima (oni dijele interijer trokuta od eksterijera); pikseli u interijeru uvijek spadaju u točno jedan trokut jer nastaju interpolacijom između dva kraja na suprotnim bridovima. Dakle, moramo osigurati da pri iscrtavanju piksela

[5] Christopher Bryan Hecker (1970.), američki programer videoigara i urednik *Journal of Graphics Tools*[8].

koji bi trebali ležati točno na bridu uvijek iscrtamo *točno jedan* piksel na toj koordinati, što znači da od dva susjedna trokuta koja dijele taj brid, *točno jedan* mora iscrtati svoj piksel i na taj način smo zagarantirali spomenutu bijekciju. Pritom moramo definirati pravila koja će rasterizator koristiti pri odlučivanju treba li iscrtati trenutni piksel na bridu ili ne. Pri ovakvoj rasterizaciji interpoliramo između suprotnih bridova trokuta koje zato za svaki trokut možemo nazvati lijevim i desnim bridovima, s tim da izdvajamo poseban slučaj kada trokut ima jedan brid koji je sasvim horizontalan – to je jedini slučaj u kojemu pri šetnji niz suprotne bridove nije nikada potrebno preći s jednog brida na drugi koji dijeli isti kraj. To znači da imamo u principu četiri izbora, komibinirana od dva „horizontalna" i dva „vertikalna" odabira rubnih piksela. U horizontalnom smislu imamo:

1. Iscrtavaj samo rubne piksele na lijevom bridu;
2. Iscrtavaj samo rubne piksele na desnom bridu.

S druge strane, vertikalno imamo:

1. Iscrtavaj samo rubne piksele na gornjem bridu – to je brid koji je sasvim horizontalan te je ujedno i onaj s najvišim y koordinatama od svih;
2. Iscrtavaj samo rubne piksele na donjem bridu.

Bilo koja kombinacija ovih dviju odabira je moguća, no mi ćemo odabrati gornju-lijevo konvenciju ispunjavanja (engl. *top-left fill convention*). To znači da lijeve bridove ne diramo i oni se iscrtavaju normalno, no desne moramo nekako manipulirati kako bi se preskočili prilikom iscrtavanja scan linija. Budući da ćemo skenirati od lijevom ka desnom bridu, to je ekvivalentno preskakanju posljednjeg piksela svake scan linije; za one linije koje su duljine ≤ 1 piksel, iscrtavanje se u potpunosti preskače i prelazimo na iduću liniju. Također, vertikalno moramo preskočiti donji brid (ako jedan takav postoji u trenutnom trokutu), što se uvijek lako može napraviti na razini glavne petlje koja šeće po suprotnim bridovima – dovoljno je samo umanjiti visinu za jedan.

Pri šetnji niz bridove trokuta, kao i pri skeniranju scan linija, moramo interpolirati atribute (poput teksturnih koordinata) uzduž bridova i preko interijera, i to na perspektivno korektan način. Objasnimo što to točno znači: svi vrhovi imaju određenu vrijednost konkretnog atributa u koje je programer postavio još u model-prostoru – tamo su svi atributiu linearni, što znači da ih se slobodno može linearno interpolirati između svih vrhova trokuta u tom 3D prostoru na sljedeći način:

$$u = \alpha u_0 + \beta u_1 + \gamma u_2, \tag{6}$$

gdje je u vrijednost atributa u točki u trokutu koja odgovara baricentričnim koordinatama (α, β, γ), s tim da znamo da je $\alpha + \beta + \gamma = 1$ pa je formula svedena na jednostavniju formu, a u_i su vrijednosti atributa u i-tom vrhu. Baricentrične koordinate se mogu izračunati rješavanjem odgovarajućeg Cramerovog sustava koji uključuje prostorne koordinate vrhova, tako da nema nikakvih problema s rješivosti i točnosti ovog računa. No problemi nastupaju nakon perspektivne transformacije i perspektivnog dijeljenja koja u svakom slučaju mora izvoditi dijeljenje sa z i možda još jednim dodatnim *konstantnim* faktorom koji je povezan s FOV-om d. To znači da, ako bismo interpolaciji pristupili na jednak način u ekranskom prostoru nakon takve projekcije, zanemarili bismo činjenicu da je obavljena nelinearna transformacija polaznih koordinata te, kao što je prije spomenuto, implementirali *afini* rasterizator (tako nazvan zato što ignorira perspektivne transformacije i točan je samo do na afine). Da bismo vidjeli što se događa pri takvoj transformaciji, uočimo da, pošto afine transformacije čuvaju afine kombinacije atributa, vrijedi opet

$$u(M(\mathbf{x})) = u = \alpha u_0 + \beta u_1 + \gamma u_2, \tag{7}$$

gdje je M afina transformacije koja uključuje sve izmnožene matrice prije same perspektivne transformacije ($u(M(\mathbf{x}))$ označava primjenu funkcije koja preslikava transformiran vrh trokuta u željenu vrijednost atributa na tom transformiranom vrhu). Nakon primjene perspektivne matrice i množenja s d (koje ignoriramo jer se radi o konstanti različitoj od nule pomnoženoj sa svim faktorima koju zato možemo

pokratiti) dobivamo

$$\frac{u}{z} = \alpha\frac{u_o}{z_0} + \beta\frac{u_1}{z_1} + \gamma\frac{u_2}{z_2}, \tag{8}$$

što znači da imamo istu relaciju kao u (6), samo što sada moramo interpolirati $\frac{u}{z}$, a ne više samo u! Drugim riječima, $\frac{u}{z}$ je linearan (u smislu linearne interpolacije) u *ekranskom* prostoru, dok je sam u linearan u bilo kojem prethodnom 3D prostoru. Na sličan način možemo vidjeti i da je $\frac{1}{z}$ uvijek linearno u ekranskom prostoru, naime za konstantnu funkciju $f(\mathbf{x}') = 1$ (gdje je $\mathbf{x}'_i$ neki vrh trokuta nakon proizvoljne afine transformacije) koja ima ulogu atributa (što znači da čuva afine kombinacije) vrijedi

$$\mathbf{x}' = \alpha\mathbf{x}'_0 + \beta\mathbf{x}'_1 + \gamma\mathbf{x}'_2 \text{ primijenimo sve transformacije do ekranskog prostora} \tag{9}$$

$$\frac{\mathbf{x}'}{z} = \alpha\frac{\mathbf{x}'_0}{z_0} + \beta\frac{\mathbf{x}'_1}{z_1} + \gamma\frac{\mathbf{x}'_2}{z_2} \text{ primijenimo funkciju} \tag{10}$$

$$f(\frac{\mathbf{x}'}{z}) = \alpha f(\frac{\mathbf{x}'_0}{z_0}) + \beta f(\frac{\mathbf{x}'_1}{z_1}) + \gamma f(\frac{\mathbf{x}'_2}{z_2}) \tag{11}$$

$$\frac{1}{z} = \frac{\alpha}{z_0} + \frac{\beta}{z_1} + \frac{\gamma}{z_2}. \tag{12}$$

Dakle, kako bismo ispravno mogli interpolirati atribute uzduž trokuta, moramo uvijek interpolirati polazni atribut podijeljen sa z tog vrha. Zatim, na svakom pikselu na kojem trebamo evaluirati taj atribut (npr. teksturne koordinate (u,v) kako bismo mogli uzorkovati teksturu na tom pikselu) izvedemo dijeljenje sa $\frac{1}{z}$ tog piksela, koji opet dobijemo zasebnom linearnom interpolacijom z atributa vrhova ($\frac{\frac{u}{z}}{\frac{1}{z}} = u$). Znači, praktički najmanji broj atributa za perspektivno-korektni rasterizator s mapiranjem tekstura jest tri: u, v (2D teksturne koordinate) i z.

Nadalje, zbog linearnosti tako podijeljenih atributa možemo intuitivno zaključiti da su parcijalne derivacije po dvije ekranske koordinate svih linearnih atributa u ekranskom prostoru konstantne uzduž cijelog trokuta. Te se parcijalne derivacije u literaturi za ove rasterizacije najčešće nazivaju *gradijentama trokuta* (engl. *triangle gradients*). Formalno, za atribut u (ovdje već smatramo da su svi oni podijeljeni s odgovarajućom z koordinatom u ekranskom prostoru, tako da to više nećemo spominjati) možemo definirati funkciju

$$u(x,y) = u_0 + \alpha(x,y)\,(u_1 - u_0) + \beta(x,y)\,(u_2 - u_0), \tag{13}$$

gdje su

$$\alpha(x,y) = \frac{(x-x_0)\,(y_2-y_0) - (x_2-x_0)\,(y-y_0)}{(x_1-x_0)\,(y_2-y_0) - (x_2-x_0)\,(y_1-y_0)}, \tag{14}$$

$$\beta(x,y) = \frac{(x_1-x_0)\,(y-y_0) - (x-x_0)\,(y_1-y_0)}{(x_1-x_0)\,(y_2-y_0) - (x_2-x_0)\,(y_1-y_0)} \tag{15}$$

dobiveni kao rješenja Cramerovog sustava za ekranski piksel $\mathbf{x} = (x,y)$ ($(x_i,y_i), i \in \{0,1,2\}$ su koordinate triju vrhova):

$$\mathbf{x} = x_0 + \alpha(x_1 - x_0) + \beta(x_2 - x_0) \tag{16}$$

$\Leftrightarrow$

$$\begin{cases} x &= x_0 + \alpha(x_1 - x_0) + \beta(x_2 - x_0) \\ y &= y_0 + \alpha(y_1 - y_0) + \beta(y_2 - y_0) \end{cases}$$

Vratimo se na (13) i izračunajmo parcijalne derivacije $\frac{\partial u}{\partial x}$ i $\frac{\partial u}{\partial y}$. Uočimo da nama ustvari ne trebaju točne derivacije nego samo njihove numeričke aproksimacije do na rezoluciju ekrana tj. do maksimalne točnosti koju trenutni diskretni raster nudi, stoga jednostavno možemo izračunati te aproksimacije $\frac{\mathrm{d}u}{\mathrm{d}x}$ i

$\frac{du}{dy}$ uzimanjem odvojenih horizontalnih odnosno vertikalnih razlika vrijednosti $u(x,y)$ od jednog piksela:

$$\frac{du}{dx} = u(x_0+1, y_0) - u(x_0, y_0) = \tag{17}$$

$$= \frac{(u_1-u_0)\,(y_2-y_0)}{(x_1-x_0)\,(y_2-y_0)-(x_2-x_0)\,(y_1-y_0)} + \frac{(u_2-u_0)\,(y_0-y_1)}{(x_1-x_0)\,(y_2-y_0)-(x_2-x_0)\,(y_1-y_0)} \tag{18}$$

$$\frac{du}{dy} = u(x_0, y_0+1) - u(x_0, y_0) = \tag{19}$$

$$= \frac{(u_1-u_0)\,(x_0-x_2)}{(x_1-x_0)\,(y_2-y_0)-(x_2-x_0)\,(y_1-y_0)} + \frac{(u_2-u_0)\,(x_1-x_0)}{(x_1-x_0)\,(y_2-y_0)-(x_2-x_0)\,(y_1-y_0)} \tag{20}$$

(17) i (19) nam omogućuju izračunavanje gradijenata trokuta *samo jedanput* za cijeli trokut u pripremnoj fazi njegovog iscrtavanja, a zatim pri iscrtavanju evaluiramo atribute jednostavnim inkrementalnim zbrajanjem/oduzimanjem vrijednosti pojedine gradijente, ovisno o tome u kome smjeru se krećemo po bridovima (kroz scan linije se uvijek krećemo slijeva na desno, dakle samo x koordinata strogo monotono raste, dok je y konstantna pa je evaluiranje atributa tu još brže). Sve operacije bi trebale biti izvedene u 16.16 aritmetici.

Slijedi jedna implementacija svega do sada opisanog u C++-u; pritom se koriste sve funkcije i metode definirane iznad.

Listing 6: Jedna moguća nepotpuna implementacija scanline rasterizatora s fiksnim zarezom u C++-u; pretpostavljamo da svaki trokut ima pristup framebufferu i z-bufferu

```
1  struct Gradijenta //aproksimacije parcijalnih derivacija nekog atributa trokuta
2  {
3    Fixed16 ddx, ddy;
4  };
6
6  struct UlazniAtribut
7  {
8    float u0, u1, u2;
9  };
11
11 struct Atribut
12 {
13   Fixed16 u0, u1, u2;
14 };
16
16 struct Slika2D
17 {
18   uint32_t* pIshodiste; //pakirani tekseli u formatu R8G8B8 linearno (R na MSB), alfa se
         ne koristi; ishodiste je u gornjem lijevom kutu
19   uint16_t sirina, visina;
20 };
22
22 class TrokutRast
23 {
24   Fixed16 x0, y0, rz0, x1, y1, rz1, x2, y2, rz2; //rzi su recipročni z-ovi
25   int brojAtributa;
26   Atribut* atributi;
27   Gradijenta* gradijente;
28   Gradijenta rz;
29   const Slika2D* tekstura; // pokazivač na teksel (centar) u gornjem lijevom kutu
30   Fixed16 x1x0, x0x2, y2y0, y0y1, rnazivnik; //ovi se izrazi kao razlike pojavljuju u (17)
         i (19), tu se cacheiraju radi optimizacije, da ih ne računamo za svaki atribut iznova
31   //uint32_t* fb, int32_t* zb, uint16_t w; // ovo su vrijednosti koje su implicitno
         dostupne ovom objektu izvana (recimo da su globalne varijable, radi jednostavnosti)
33
33   TrokutRast(int nAtributa, Atribut* pAtributi, Gradijenta* pGradijente, Gradijenta& RZ,
         const Slika2D* tex, Fixed16 X0, Fixed16 Y0, Fixed16 RZ0, Fixed16 X1, Fixed16 Y1,
         Fixed16 RZ1, Fixed16 X2, Fixed16 Y2, Fixed16 RZ2) : x0(X0), y0(Y0), rz0(RZ0), x1(X1
         ), y1(Y1), rz1(RZ1), x2(X2), y2(Y2), rz2(RZ2), brojAtributa(nAtributa), atributi(
         pAtributi), gradijente(pGradijente), rz(RZ), tekstura(tex) {}
```

```
35
35    void IzracunajGradijentu(Gradijenta& grad, const Fixed16* pU)
36    {
37      Fixed16 u0 = *(pU++);
38      Fixed16 u1 = *(pU++);
39      Fixed16 u2 = *pU;
40      grad.ddx = ((u1-u0)*y2y0 + (u2-u0)*y0y1) * rnazivnik;
41      grad.ddy = ((u1-u0)*x0x2 + (u2-u0)*x1x0) * rnazivnik;
42    }
44
44    void PripremiTrokut(const Vektor3D& A, const Vektor3D& B, const Vektor3D& C, const
          UlazniAtribut* ulazi) //trokut se konstruira iz 2D ekranskih koordinata, gdje se
          dodan z smatra pseudo-udaljenošću i koristi se samo za z-buffering
45    {
46      //pretvori koordinate u Fixed16
47      Fixed16* pKoordinata = &x0;
48      for(auto& vrh : {A, B, C})
49      {
50        for(float koord : {vrh.x, vrh.y})
51        {
52          *(pKoordinata++) = koord;
53        }
54        *(pKoordinata++) = static_cast<Fixed16>(vrh.z).Reciproc();
55      }
57
57      //cacheiraj podizraze
58      x1x0 = x1 - x0;
59      x0x2 = x0 - x2;
60      y2y0 = y2 - y0;
61      y0y1 = y0 - y1;
62      rnazivnik = (x1x0 * y2y0 - x0x2 * y0y1);
63      rnazivnik.Reciproc();
65
65      //pretvori atribute
66      atributi = new Atribut[brojAtributa]; //[6]
67      gradijente = new Gradijenta[brojAtributa];
69
69      constexpr std::size_t razmak = alignof(Atribut)-1;
70      constexpr std::size_t razmakUlaza = alignof(UlazniAtribut)-1;
71      Fixed16 rzi[] = {rz0,rz1,rz2};
72      Fixed16* pAtribut = &(atributi[0].u0);
73      float* pUi = &ulazi[0].u0;
74      for(int i = 0; i < brojAtributa; ++i)
75      {
76        //int j = 0;
78        //for(Fixed16 ui : {ulazi[i].u0, ulazi[i].u1, ulazi[i].u2})
79        for(int j = 0; j < 3; ++j)
80        {
81          *(pAtribut++) = *(pUi++) * rzi[j];
82          //*(pAtribut++) = ui * rzi[j++];
83        }
84        IzracunajGradijentu(&gradijente[i], pAtribut-3);//, *(pAtribut-2), *(pAtribut-1));
86
86        char* pBajtAtributa = reinterpret_cast<char*>(pAtribut);
87        pBajtAtributa += razmak;
88        pAtribut = reinterpret_cast<Fixed16*>(pBajtAtributa);
90
90        pBajtAtributa = reinterpret_cast<char*>(pUi);
91        pBajtAtributa += razmakUlaza;
92        pUi = reinterpret_cast<float*>(pBajtAtributa);
94
94      }
96
96      IzracunajGradijentu(&rz, rzi);
97    }
99
99    Fixed16 EvaluirajAtribut(Fixed16 atribut, Fixed16 trenutniZ)
```

[6] Ovdje smo izostavili implementaciju memorijskih bazena koja omogućava da se preko operatora `new` ne rade uvijek nove (relativno sitne) alokacije potrebne za atribute i sl. već da se sva ta memorija uzima iz jednog velikog predalociranog bloka (idealno veličine koja je višekratnik duljine stranice), što poboljšati performanse i smanjiti fragmentaciju memorije, pošto mi skoro točno znamo koliko je memorije potrebno za iscrtavanje scene s N trokutova i nije potrebno raditi opće alokacije.

```
100   {
101     return atribut / trenutniZ;
102   }
104
104   void NacrtajScanliniju(Fixed16* scanlineAtributi, Fixed16 lijeviX, Fixed16 desniX,
        Fixed16 y, Fixed16 lijeviZ)
105   {
106     Fixed16 yKorekcija, xLijevaKorekcija;//, xDesnaKorekcija;
107     yKorekcija.broj = y.broj & 0xffff;
108     xLijevaKorekcija.broj = lijeviX.broj & 0xffff;
109     //xDesnaKorekcija = desniX.broj & 0xffff;
111
111     for(int i = 0; i < brojAtributa; ++i)
112     {
113       scanlineAtributi[i] -= (gradijente[i].ddy*yKorekcija + gradijente[i].ddx*
            xLijevaKorekcija);
114 //      desniAtributi[i] -= (gradijente[i].ddy*yKorekcija + gradijente[i].ddx*
      xDesnaKorekcija);
115     }
116     lijeviZ -= (rz.ddy*yKorekcija + rz.ddx*xLijevaKorekcija);
117 //    desniZ -= (rz.ddy*yKorekcija + rz.ddx*xDesnaKorekcija);
119
119     uint16_t lijeviCentar = lijeviX.Trunc_Spec();
120     uint16_t desniCentar = desniX.Trunc_spec();
121     uint16_t yCentar = y.Trunc_Spec();
122     uint16_t sirina = desniCentar - lijeviCentar;
123     uint32_t* pix = (fb+w*yCentar+lijeviCentar); //ovo se može izbjeći pomicanjem
            pokazivača zajedno sa scanlinijama i uz lijevi brid
125
125     for(uint16_t i = 0; i < sirina; ++i)
126     {
127       Fixed16 evaluiraniAtribut = EvaluirajAtribut(scanlineAtributi[nekiAtribut], lijeviZ
            );
128       //tu bi sada išlo dubinsko testiranje pomoću z-buffera te na kraju, ako je ono
            uspješno, iscrtavanje trenutnog piksela; ako ne, vraćamo se iz funkcije;
            detalji će biti obrađeni u 3.4
129       *(pix++) = /*iskorištavanje atributa na neki način...*/
130       for(int j = 0; j < brojAtributa; ++j)
131       {
132         scanlineAtributi[j] += gradijente[j].ddx;
133         lijeviZ += rz.ddx;
134       }
135     }
136   }
138
138   void UcitajAtributeBrida(int index, Fixed16* pDest) //učitava atribute početnog vrha
        brida indeksa index kako bi se mogli interpolirati po tom bridu; ovo je moguće
        izbjeći uz drukčiji format atributa, vidjeti komentar ispod
139   {
140     const Atribut* pAtributi;
141     for(int i = 0; i < brojAtributa; ++i)
142     {
143       pAtributi = &atributi[i];
144       const Fixed16* pKomponenta = reinterpret_cast<const Fixed16*>(pAtributi);
145       //pAtributi = static_cast<Fixed16*>(pAtributi);
146       *(pDest++) = *(pKomponenta+index);
147     }
148   }
150
150   void NacrtajJednostavanTrokut(uint16_t visina, Fixed16 lijeviX, Fixed16 desniX, Fixed16
        y, Fixed16 lijevaKosina, Fixed16 desnaKosina, int lijeviIndex ) //trokut s gornjim
        horizontalnim bridom
151   {
152     Fixed16* lijeviAtributi = new Fixed16[brojAtributa]; //ova se alokacija mogla u
          potpunosti izbjeći da smo koristili pohranu atributa u niz u formatu [
          u0v0w0u1v1w1...] koji bi nam omogućio da jednostavno pročitamo sve atribute u
          nizu za onaj vrh koji nam treba
153     Fixed16* scanlineAtributi = new Fixed16[brojAtributa];
154     Fixed16* pKoord = &rz0;
155     Fixed16 lijeviZ = *(pKoord+lijeviIndex*3);//, desniZ = *(pKoord+2+desniIndex*3);
157
157     UcitajAtributeBrida(lijeviIndex, lijeviAtributi);
158     //UcitajAtributeBrida(lijeviIndex, scanlineAtributi); //tu bi se mogao koristiti i
```

```
        memcpy() iz lijevih atributa
159     for(; visina > 0; --visina,--y)
160     {
161       std::memcpy(scanlineAtributi, lijeviAtributi, brojAtributa*sizeof(Fixed16));
162       NacrtajScanliniju(scanlineAtributi, lijeviX, desniX, y, lijeviZ);
163       lijeviX -= lijevaKosina;
164       desniX -= desnaKosina;
165       for(int i = 0; i < brojAtributa; ++i)
166       {
167         lijeviAtributi[i] -= (gradijente[i].ddy + gradijente[i].ddx*lijevaKosina);
168       }
169       lijeviZ -= (rz.ddy + rz.ddx*lijevaKosina);
171
171     }
173
173     delete[] scanlineAtributi;
174     delete[] lijeviAtributi;
175   }
177
177   void NacrtajOpcenitTrokut(const Vektor2D& najvisi, const Vektor2D& srednji, const
        Vektor2D& najnizi, int indexVrha)
178   {
179     Fixed16* lijeviAtributi = new Fixed16[brojAtributa];
180     Fixed16* scanlineAtributi = new Fixed16[brojAtributa];
181     Fixed16* pKoord = &rz0;
182     Fixed16 lijeviZ = *(pKoord+indexVrha*3);
184
184     UcitajAtributeBrida(indexVrha, lijeviAtributi);
186
186     Fixed16 y = najvisi->y;
187     uint16_t doSrednjeg = static_cast<uint16_t>(najvisi->y) - static_cast<uint16_t>(
          srednji->y);
188     Fixed16 lijeviX, desniX;
189     lijeviX = desniX = najvisi->x;
190     Fixed16 rdy = najvisi->y - srednji->y;
191     rdy.Reciproc();
193
193     Fixed16 xDonjiLijevi, xDonjiDesni;
194     bool nastavakLijevo;
195     if(srednji->x < najnizi->x)
196     {
197       xDonjiLijevi = srednji->x;
198       xDonjiDesni = najnizi->x;
199       nastavakLijevo = false;
200     }
202
202     else
203     {
204       xDonjiLijevi = najnizi->x;
205       xDonjiDesni = srednji->x;
206       nastavakLijevo = true;
207     }
209
209     Fixed16 lijevaKosina = (najvisi->x - xDonjiLijevi) * rdy;
210     Fixed16 desnaKosina = (najvisi->x - xDonjiDesni) * rdy;
211      Fixed16 trecaKosina = (srednji->x - najnizi->x) / (srednji->y - najnizi->y); //ovo
           će se koristiti za jednostavan trokut kada dođemo do sredine
212     for(; doSrednjeg > 0; --doSrednjeg,--y)
213     {
214       std::memcpy(scanlineAtributi, lijeviAtributi, brojAtributa*sizeof(Fixed16));
215       NacrtajScanliniju(scanlineAtributi, lijeviX, desniX, y, lijeviZ);
216       lijeviX -= lijevaKosina;
217       desniX -= desnaKosina;
218       for(int i = 0; i < brojAtributa; ++i)
219       {
220         lijeviAtributi[i] -= (gradijente[i].ddy + gradijente[i].ddx*lijevaKosina);
221       }
222       lijeviZ -= (rz.ddy + rz.ddx*lijevaKosina);
223     }
225
225     uint16_t preostalaVisina = static_cast<uint16_t>(srednji->y) - static_cast<uint16_t>(
          najnizi->y);
226     if(nastavakLijevo)
```

```
227         desnaKosina = trecaKosina;
228       else
229         lijevaKosina = trecaKosina;
230       //ovo je isti kod kao za crtanje jednostavnog trokuta, ali da izbjegnemo stvaranje
              novog objekta, ovdje kopiramo i prilagođavamo kod; vidjeti diskusiju u tekstu
              ispod
231       for(; preostalaVisina > 0; --preostalaVisina,--y)
232       {
233         std::memcpy(scanlineAtributi, lijeviAtributi, brojAtributa*sizeof(Fixed16));
234         NacrtajScanliniju(scanlineAtributi, lijeviX, desniX, y, lijeviZ);
235         lijeviX -= lijevaKosina;
236         desniX -= desnaKosina;
237         for(int i = 0; i < brojAtributa; ++i)
238         {
239           lijeviAtributi[i] -= (gradijente[i].ddy + gradijente[i].ddx*lijevaKosina);
240         }
241         lijeviZ -= (rz.ddy + rz.ddx*lijevaKosina);
243
243       }
245
245       delete[] scanlineAtributi;
246       delete[] lijeviAtributi;
247     }
250
250
250     public:
251     TrokutRast(const Vektor3D& A, const Vektor3D& B, const Vektor3D& C, int n, const
            UlazniAtribut* ulazi, const Slika2D* tex) : brojAtributa(n), tekstura(tex)//, fb(
            pFramebuffer), zb(pZbuffer), w(sirina)
252     {
253       PripremiTrokut(A, B, C, ulazi);
254     }
256
256     ~TrokutRast()
257     {
258       delete[] atributi;
259       delete[] gradijente;
260     }
262
262     void Iscrtaj()//fb i zb je pokazivač na donji lijevi piksel kolor-framebuffera odnosno
            z-buffera; format fb-a je isti kao za teksture
263     {
264   //    Vektor2D v0(x0, y0); //Vektor2D je ovdje specijaliziran za Fixed16 koordinate
265   //    Vektor2D v1(x1, y1);
266   //    Vektor2D v2(x2, y2);
267       Vektor2D v[] = {Vektor2D(x0, y0), Vektor2D(x1, y1), Vektor2D(x2, y2)};
268       // const Vektor2D *najvisi, *najnizi, *srednji;
269       int najvisi, najnizi, srednji; //indeksi u niz vrhova 'v'
271
271       if(y0 >= y1)  //zamijeniti sve ovo sa std::minmax()...
272       {
273         if(y0 >= y2)
274         {
275           najvisi = 0;
276           if(y2 <= y1)
277           {
278             srednji = 1;
279             najnizi = 2;
280           }
281           else
282           {
283             srednji = 2;
284             najnizi = 1;
285           }
286         }
288
288         else
289         {
290           najvisi = 2;
291           najnizi = 1;
292           srednji = 0;
293         }
294       }
```

```
295       else
296       {
297         if(y1 <= y2)
298         {
299           najvisi = 2;
300           najnizi = 0;
301           srednji = 1;
302         }
303         else
304         {
305           najvisi = 1;
306           if(y0 <= y2)
307           {
308             najnizi = 0;
309             srednji = 2;
310           }
311           else
312           {
313             srednji = 0;
314             najnizi = 2;
315           }
316         }
317       }
320
320
320       if(v[najvisi].y == v[srednji].y)
321       {
322         const Vektor2D *lijeviVrh, *desniVrh;
323         int lijeviIndex;
324         if(v[najvisi].x < v[srednji].x)
325         {
326           lijeviVrh = &v[najvisi];
327           desniVrh = &v[srednji];
328           lijeviIndex = najvisi;
329         }
330         else
331         {
332           lijeviVrh = &v[srednji];
333           desniVrh = &v[najvisi];
334           lijeviIndex = srednji;
335         }
336         Fixed16 lijeviX = v[lijeviVrh].x;
337         Fixed16 desniX = v[desniVrh].x;
339
339         uint16_t visinaPix = static_cast<uint16_t>(v[najvisi].y) - static_cast<uint16_t>(v[
                najnizi].y);
340         Fixed16 visina = v[najvisi].y - v[najnizi].y;
341         Fixed16 rvisina = visina.Reciproc();
342         Fixed16 lijevaKosina = (lijeviX - najnizi->x) * rvisina;
343         Fixed16 desnaKosina = (desniX - najnizi->x) * rvisina;
345
345         NacrtajJednostavanTrokut(visinaPix, lijeviX, desniX, v[najvisi].y, lijevaKosina,
                desnaKosina, lijeviIndex);
346       }
348
348       //if(srednji->y == najnizi->y)
349       //  --visina;
350       else
351         NacrtajOpcenitTrokut(v[najvisi], v[srednji], v[najnizi], najvisi);
352     }
353 };
```

U gornjem kodu ima puno implementacijskih detalja, no ono što smjesta valja naglasiti jest da je ova implementacija specifično napravljena za demonstraciju određenih svojstava algoritma te su zanemarene neke mogućnosti optimizacije, npr. vidimo da funkcija za crtanje „općenitih" trokutova 177 ustvari sadrži dio koda upravo iz funkcije 150 za crtanje *samo* takvih trokutova, što znači da se ove funkcije moglo spojiti u jedno što se i čini kod ovakvih rasterizatora – ovdje radi demonstrativnih razloga ostavljamo ta dva slučaja razdvojenim tako da čitatelj može prvo razmišljati o jednostavnijem slučaju, pa zatim vidjeti kako sve zajedno s par dodataka funkcionira u općem slučaju. Drugim riječima, vidimo da opći

trokut uvijek sadrži jedan jednostavan podtrokut, gdje „jednostavnost" označava posjedovanje gornjeg horizontalnog brida. Nadalje, uočimo potrebu ovog algoritma u liniji 108 da za svaku scanliniju korigira interpolirane x i y koordinate bridova; to je potrebno zato što šetnja niz bridove ne garantira da ćemo sletiti točno na centar piksela, dapače to je vrlo rijetka pojava (kosine su većinom necjelobrojne, krajevi bridova imaju većinom necjelobrojne koordinate). Stoga je potrebno za svaku tako generiranu scanliniju odbaciti decimalni dio (radimo najveće-cijelo-od koordinata koje su uvijek pozitivne) od x i y i korigirati *sve atribute* da odgovaraju toj novoj cjelobrojnoj koordinati – proces poznat kao *prestepping*. Taj detalj se možda ne čini značajnim, no nakon što se prijeđe određen prag broja atributa, ovo počinje imati negativan utjecaj na performanse. Postoji alternativan pristup ovom algoritmu koji skoro sasvim uklanja potrebu za *prestepping*-om, a kojeg smo već natuknuli kad smo govorili o Bresenhamovom algoritmu. Naime, ako računanje kosina i spuštanje niz bridove uspomoć njih zamijenimo sa 4 pristupom interpolaciji (gdje bi se u petlji u 1 isključivo iteriralo po y koordinati) između dva para vrhova, s time da se ažuriranje atributa i dalje obavlja i za x pomake manje od cijelog piksela (više o tome će biti riječi kasnije), a ne samo kada se prijeđe `prag`, tada imamo puno efikasniju verziju ovog algoritma, barem u slučajevima s puno atributa koje sada više nije nužno korigirati za svaku liniju. Primijetimo da i dalje jest potrebno obaviti jednu jedinu inicijalnu korekciju, i to onu koja smješta najviši vrh trokuta u centar najbližeg piksela, korigira njegove atribute i zatim napravi isto za preostala dva vrha ali *bez korekcije atributa* – nakon toga, za generiranje svih scanlinija, više nisu potrebne nikakve korekcije[7]. Ovaj pristup također omogućava razne mikrooptimizacije poput izbjegavanja 16.16 dijeljenja i množenja za koordinate, koje se tada koristi samo za interpolaciju i konačnu evaluaciju *atributa* po scanlinijama odnosno pikselima.

Ovaj algoritam ima očit nedostatak u tome što zahtijeva alokaciju barem jednog dodatnog međuspremnika za interpolirane atribute na trenutnoj scanliniji. Naime, budući da je potrebno spustiti se niz čitavi lijevi brid radi generiranja svih potrebnih scanlinija, treba nam i međuspremnik za atribute po tim scanlinijama koji je različit od onog za atribute na trenutnom pikselu na lijevom bridu, jer se spuštanje niz brid mora nastaviti i nakon iscrtavanja trenutne scanlinije s nepromijenjenim lijevim atributima. Tu se još može javiti i potreba za dodatnim međuspremnikom, kao u ovoj implementaciji, koji drži te atribute trenutne točke na lijevom bridu, ako format pohrane ulaznih i pretvorenih atributa nije ispresjecan na način da se prvo pohranjuju *svi* atributi jednog, pa zatim svi atributi drugog i na kraju svi atributi posljednjeg vrha. Takav format bi omogućio direktno iščitavanje onog podniza koji odgovara najvišem vrhu trokuta, te njegovu interpolaciju (u smislu svih atributa koji su u tom nizu) niz lijevi brid bez potrebe za dodatnom alokacijom (osim one za scanline atribute koja je nužna).

Čak i sa svim spomenutim optimizacijama, ovaj algoritam pati od još nekoliko nedostataka:

- Neefikasno iskorištavanje cachea – unutar jednog trokuta, cache se koristi poprilično konzistentno, no budući da se trokuti iscrtavaju u praktički nasumičnom redoslijedu s različitim teksturama i atributima te na nepovezanim lokacijama unutar framebuffera, podatkovni cache počinje patiti pri imalo složenijim scenama;
- Agnostičnost na veličinu trokuta – bez obzira koja je veličina trokuta, ovaj algoritam uvijek izvodi isti kôd, što znači da bilo kakva podjela posla na razini trokutova može završiti s vrlo nepredvidivim vremenima pojedinačnih izvođenja jedinica konkurentnosti, što se odmah odražava na sljedeći nedostatak; štoviše, ovaj algoritam sprječava bilo kakav brz podijeli-pa-zavladaj hijerarhijski pristup podjele framebuffera na regije nezavisne rasterizacije jer bi on zahtijevao stvaranje regija koje režu barem neke trokutove kroz barem jedan brid;
- Nemogućnost općenite paralelizacije – osim iznad navedenih problema koji utječu na paraleliza-

[7] Naglasimo da, bez obzira što u ovom pristupu koristimo Bresenhamov algoritam za spuštanje niz bridove što znači da nam za x i y koordinate striktno ne trebaju kosine, i dalje ih jest potrebno izračunati u 16.16 aritmetici kao iznad i koristiti isključivo za interpolaciju *atributa* – naime, atribute možemo ažurirati i prije nego što je prijeđen prag, tj. prije nego što se dogodio skok x koordinate na sljedeću, jer nam to omogućava ekvivalelntan rezultat kao u 6 gdje se to dobilo npr. linijom 167. Ukoliko bismo to zanemarili učiniti, dobili bismo rasterizator u kojima teksture mapirane na trokutove izgledaju kao da trzaju i skaču uokolo po pikselima na neki način, što je upravo rezultat nedostatka ove tzv. potpikselne preciznosti interpolacije (engl. *subpixel precision*).

ciju, postoji još i problem z-buffera (i alfa miješanja boja, no time se ovdje ne bavimo): kada bi više niti konkurentno izvodilo iscrtavanje trokutova, postavlilo bi se pitanje treba li trenutna nit zapisati generirani piksel na njegovu lokaciju ili ne, jer to ovisi o rezultatu dubinskog testiranja pomoću z-buffera, koji pak ovisi o tome jesu li svi ostali trokutovi (koji okupiraju isti piksel na nekoj drugoj dubini) zapisali svoje vrijednosti u z-buffer; u svakom slučaju, krajnji će rezultat biti točan (pod uvjetom da dubinsko testiranje radi korektno), no možda će zahtijevati veći broj nepotrebno zapisanih piksela na jednu te istu lokaciju framebuffera od različitih trokutova, od kojih je na kraju samo jedan „pobijedio"; ovo također pretpostavlja da se zapisivanje piksela od strane različitih niti u zajednički framebuffer događa s ispravnim zaključavanjem tj. sinkronizacijskim mehanizmima, koje može značajno smanjiti performanse; to se donekle može ublažiti tako da se pokuša napraviti gore spomenute nezavisne regije takvima da u njih spadaju trokuti koji okupiraju otprilike jednaku površinu na različitim dubinama, no to je sam po sebi vrlo zahtjevan postupak dodatno zakompliciran gore spomenutim problemima.

Vidimo da zbog navedenih razloga ovaj algoritam rasterizacije ne može biti preferiran u današnje doba masivne paralelnosti i visoke osjetljivosti na iskorištenje cachea, no on jest bio vrlo važan i korišten 1990-ih prvenstveno zbog toga što je omogućavao nešto što sljedeći tip rasterizatora koji ćemo obraditi ne podržava – perspektivno *kvazi-korektnu* interpolaciju atributa. Naime, ovaj algoritam omogućava da se u funkciji 104 interpolacija svih atributa [8]) odvija na način da se prvo scanlinija subdivira tako da se smjeste ekvidistantne točke (na cjelobrojnim piksel koordinatama) $P_1, \ldots, P_{n-1}$ (P_0 i P_n su implicitno lijevi odnosno desni kraj scanlinije koje već znamo) na kojima se atributi evaluiraju na uobičajen, potpuno perspektivno-korektan način, dok se zatim atributi na svim pikselima između P_i i $P_{i+1}, i \in \{0, 1, \ldots, n\}$ dobiju linearnom interpolacijom između već izračunatih atributa na P_i i P_{i+1}. Očito, takva linearna interpolacija nije matematički ekvivalentna onome što dobivamo kod standardne implementacije, jer u potpunosti zanemaruje nelinearan utjecaj koji ima $\frac{1}{z}$ za piksele između P_i-ova, no greške koje njome nastaju, uz ispravno odabranu udaljenost točaka P_i, su izrazito male i tipično se u to doba nisu mogle uočiti u realnom vremenu. Ako je D udaljenost dvaju susjednih P_i-ova, tada sa $D \rightsquigarrow 0$ dobivamo perspektivno-korektnu interpolaciju, ili drugim riječima, ovaj postupak ustvari linearno aproksimira hiperbolu $\frac{1}{z}$, pa možemo zaključiti da što manje $\frac{1}{z}$ gradijente vode to točnijih rezultata. To se najlakše može osigurati tako da 3D aplikacija renderira sve primitive isključivo po linijama gdje je z konstantan (engl. *lines of constant z*), pa to svojstvo uvijek trivijalno vrijedi i imamo sasvim korektnu rasterizaciju bez ikakve potrebe za relativno skupim dijeljenjem svih atributa sa z (dakle, tu nije potrebno ni smještanje gore spomenutih točaka!). Ipak, takvo je rješenje ekstremno te ga nije moguće efikasno implementirati u općenitim 3D aplikacijama, već se ono koristilo u posebno-dizajniranim iskustvima s tim svojstvom na umu. Možda i najpoznatiji primjer takve tehnike rasterizacije je videoigra DOOM (id Software, 1993) koja se renderirala isključivo po horizontalnim (podovi i stropovi) i vertikalnim (zidovi, sprajtovi) linijama gdje je z konstantan prema originalnoj zamisli Johna Carmacka[9]. Konkretna implementacija opće metode smještanja P_i-ova ne radi u gore spomenutoj sekvenci već tipično postavi dvije točke P_i i P_{i+1} (na početku je prva implicitno prisutna) te, dok se linearna interpolacija između njih događa koristeći isključivo 16.16 aritmetiku, obavlja postavljanje iduće točke P_{i+2} koristeći FPU tj. kôd u 104 koji radi nad `float` varijablama. Ovo kod superskalarnih procesora omogućava da se ALU i FPU operacije odvijaju praktički paralelno (tzv. preklapanje izvođenja instrukcija), ako je sve optimizirano do najniže razine (posebno je važan odabir D-a koji će ovisiti o dubinama cjevovoda procesora) i za specifično poznatu mikroarhitekturu te da operacija iscrtavanja podskupa scanlinije određene duljine traje skoro uvijek isti, minimalan broj ciklusa.[10] Opet, jedna od najvažnijih implementacija ovakvog rasterizatora, anticipira-

[8] Ovdje poglavito mislimo na teksturne koordinate koje su praktički bili jedini atributi uz z u to doba, što anulira spomenut nedostatak oko potrebne alokacije memorije jer se nije imalo što alocirati, samo prenositi fiksni broj varijabli preko stoga ili registara.

[9] John D. Carmack (1970.). Američki programer, poznat po brojnim inovacijama u tehnologiji renderiranja u realnom vremenu, poglavito za videoigre.

[10] Primijetimo da odabir D-a znači da duljina scanlinije možda nije djeljiva s tim brojem; u tom slučaju se posljednji segment definira s duljinom ostatka tog dijeljenja koja je jasno manja od D te stoga ne može dovesti do većih grešaka od onih koje bismo imali da su sve točke ekvidistantne.

nog još u Heckerovim člancima[7], dolazi iz id Software-a: igra Quake (id Software, 1996) je koristila ovaj tip rasterizatora kojeg je (uz brojne druge akceleracijske strukture i odluke o kojima će biti riječi u 4.3) osmislio John Carmack, a za Pentium u x86 asembleru rigorozno optimizirao Michael Abrash[11], koristeći gore spomenutu tehniku.

3.4 Hijerarhijski rasterizator jednadžbi bridova

Posljednji tip rasterizatora kojim ćemo se baviti jest ujedno i najnoviji od svih obrađenih, barem u smislu znanstvenih radova koji su ga prvi detaljno opisivali. On je postao popularan otprilike na prijelazu tisućljeća kada je postalo jasno da prije spomenute restrikcije ne omogućuju jako veliku skalabilnost i ekonomičnost pri proizvodnji vrlo specijaliziranog 3D grafičkog hardvera (još uvijek samo fiksne funkcionalnosti) koji je u to doba postajao očekivani dio svakog imalo sposobnijeg PC-a. Kao što smo natuknuli u 3.3.3, rješenje za modernije arhitekture leži u hijerarhijskom pristupu rasterizaciji, gdje bi trebalo biti moguće rekurzivno podijeliti framebuffer na regije unutar kojih se zbiva rasterizacija i koje su dio hijerarhije koja garantira da se što ranije u procesu može preskočiti renderiranje nekog trokuta ako se detektira da on ne leži na trenutnoj razini hijerarhije (jer onda nije ni na nižima) te idealno nudi visoku iskoristivost cachea. To sada već počinje ličiti na klasičan problem u grafici gdje se mora odbaciti što veći broj objekata što efikasnije kako bi se na skupo procesiranje predalo samo one koji zaista doprinose konačnim pikselima, samo što je to ovdje fundamentalan dio rasterizatora (u 2D) i stoga mora biti vrlo efikasan. Postoje različiti prisupi ovom problemu, od raznih stablastih struktura, preko BSP-a (engl. *Binary Space Partitioning*) do ravninskih krivulja. Upravo je posljednji pristup onaj koji ćemo mi opisati i implementirati u softveru, uzimajući kao osnovu rad [10]. Konkretno, koristimo aproksimaciju Hilbertove krivulje, koja fraktalno ispunjuje bilo koju ravninu, kako bismo inkrementalno dijelili framebuffer na sve manje i manje regije dok ne dođemo do nivoa piksela. Rekurzivna definicija i izgradnja Hilbertove krivulje nam omogućavaju vrlo efikasno rano odbacivanje trokutova koji su van trenutne regije, i tako sve do piksel razine. Činjenica da Hilbertova krivulja neprekidno povezuje sve piksele framebuffera te da, kada krenemo od početne točke, sigurno možemo doći do kraja šetajući niz Hilbertovu krivulju kroz sve susjedne piksele, garantira vrlo dobro iskorištavanje cachea. Usto ćemo vidjeti i da na kraju dobivena struktura (dobivena takvim rekurizvnim dijeljenjem framebuffera) predstavlja jedan poseban tip stabla.

Prvo valja reći da je ovaj tip rasterizatora jedan poseban oblik tzv. rasterizatora jednadžbi bridova trokuta. Drugim riječima, za razliku od pristupa u 3.3, ovdje nećemo procesirati trokut-po-trokut, uvijek čekajući da u potpunosti ne nacrtamo jedan dok ne pređemo na idući, već ćemo (samo figurativno, točan pristup ovog konkretnog rasterizatora je drukčiji!) „gađati" horizontalne zrake redom od vrha prema dnu ekrana te iscrtati sve piksele uzduž svake od ispaljenih zraka koji pripadaju trokutovima, što otkrivamo pomoću njihovih jednadžbi bridova. Očito, ovaj pristup je smjesta bolji sa stajališta paralelizacije jer bi svaka zraka ili veća grupa zraka mogla biti procesirana nezavisno i nema nikakve potrebe za zaključavanjem framebuffera. No i dalje postoje problemi s cacheom zbog nepredvidivosti sjecanja zraka s trokutovima i s ponavljanjem računa koje različite zrake moraju napraviti kada siječu isti trokut (radi evaluacije atributa na svim svojim sjecištima): gradijente iz (17) bi se mogle koristiti u ovom tipu rasterizatora samo kada bi svaka zraka isključivo morala presjecati samo cijele trokutove tj. i lijevi i desni brid (zraka mora i ući i izaći iz trokuta) jer jedino tada imamo sjecišta na rubovima koje nakon evaluacije možemo interpolairati po praktički nastaloj scanliniji koristeći gradijente. No taj pristup ne funkcionira ako želimo općenito gađati segmente koje nemaju širinu framebuffera, jer tada nema ni garancije da će svaki segment i ući i izaći iz trokutova koje siječe. Sve su to problemi zbog kojih se ovakav naivni pristup tipično ne koristi, već se uvijek ide na dublje hijerarhijsko rješenje. Pritom, ne gađamo nikakve zrake nego, u našem konkretnom slučaju, pratimo rekurizvni spust niz Hilbertovu krivulju od najvišeg nivoa (cijeli framebuffer) do piksel nivoa, prateći samo striktno horizontalne (gore, dolje) i vertikalne

[11] Michael Abrash, američki programer, inženjer i autor brojnih tehničkih članaka te knjige *Graphics Programming Black Book*, poznat po svojoj ekspertizi u optimizaciji softvera. Abrash je radio na Intel Larrabee arhitekturi koja je danas dio https://www.tomshardware.com/reviews/intel-larrabee-graphics,2253-2.html

(lijevo, desno) pomake po pikselima, što na kraju vizualno izgleda kao fraktalni labirint. Na piksel nivou moramo svaki put provjeriti je li piksel na trenutnoj (x,y) koordinati unutar nekog od trokuta koji do sada nisu izbačeni pri hijerarhijskoj provjeri; ako jest i ako dubinsko testiranje prođe pozitivno, iscrtamo ga pomoću perspektivno-korektno interpoliranih (u,v) teksturnih koordinata i unesemo njegovu interpoliranu z koordinatu u z-buffer. Dakle, nećemo *uopće* predračunati gradijente svih trokutova u sceni, već evaluaciju atributa radimo na drugi način opisan ispod. Ovaj rasterizator je također relativno lako paralelizirati na efikasan način, sve dok se framebuffer može u potpunosti podijeliti u n kvadratnih zona dimenzija $2^N \times 2^N$, gdje je N proizvoljni prirodni broj, zbog Hilbertove rekurzivne podijele koju objašnjavamo ispod. Ukoliko framebuffer nije takvih dimenzija da je to moguće učiniti (većina rezolucija su takve), tada imaginarno proširimo framebuffer do najbližeg većeg takvog kvadrata i njega dijelimo na podjednak način, s tim da se prazan prostor u potpunosti preskače pri šetnji po Hilbertovoj krivulji – na taj način je barem pod svim rezolucijama moguće izvoditi rasterizaciju.

Recimo i to da u ovom tipu rasterizatora ne vrijedi nužno niti jedna pretpostavka iz prošlih tipova, osim svojstava 3.3 koja su i dalje ključna. Posebno, kod ovog tipa rasterizatora pretpostavljamo da su ulazne koordinate iz projektivne ravnine, tj. homogene koordinate tipa (xw, yw, w), gdje w nije nužno 1 i gdje se već primijenila transformacija (1) (one su inicijalno `float` tipa) – drugim riječima, ulazne koordinate vrhova dobivamo prije njihova perspektivnog dijeljenja. Nadalje, ovdje ne radimo eksplicitno obrezivanje primitiva, tako da praktički postoji samo jedna glavna operacija u ovom stadiju: konstrukcija i obilaženje cijele Hilbertove krivulje, pri čemu se za atribute i za jednadžbe bridova koristi isključivo 16.16 aritmetika s implicitnim obrezivanjem i uklanjanjem potpuno vanjskih trokutova *tijekom* samog procesa (a ne prije kao u prošlom tipu rasterizatora). Dakle, *ne* očekujemo da su primitive obrezane i da ne „vire" izvan ekrana – sve to je inicijalno dozvoljeno, samo je potrebno da su trokutovi odrezani u odnosu na $z-near$ i $z-far$ ravnine.

3.4.1 Testiranje pripadnosti piksela trokutu preko jednadžbi bridova

Neka su v_0, v_1 i v_2 vrhovi nekog trokuta, gdje je za $i \in \{0,1,2\}$

$$v_i = \begin{bmatrix} x_i w_i \\ y_i w_i \\ w_i \end{bmatrix}. \tag{21}$$

Ove vektore možemo sabrati u jednu matricu trokuta $\mathbf{M}$:

$$\mathbf{M} = \begin{bmatrix} x_0 w_0 & x_1 w_1 & x_2 w_2 \\ y_0 w_0 & y_1 w_1 & y_2 w_2 \\ w_0 & w_1 & w_2 \end{bmatrix} \tag{22}$$

Nama za odlučivanje pripada li trenutni ekranski piksel $(x,y,1)$ trokutu ili ne trebaju jednadžbe normala njegovih bridova; slično kao što smo to učinili u 3.3.2, ovdje ćemo testirati predznak skalarnog produkta $(x,y,1)$ s tim normalama i prema tome donijeti zaključak. Uočimo da je pravac na kojem leži brid u potpunosti definiran s dva svoja kraja u homogenim koordinatama, a budući su u projektivnoj ravnini pravci dvodimenzionalni linearni potprostori od $\mathbb{R}^3$, on se može interpretirati i kao (hiper)ravnina u tom 3D prostoru, kroz ishodište. Ta je ravnina jednoznačno definirana s trima točkama: ishodištem te dva kraja brida. To nam je dovoljno za izračunati normale brida u 3D prostoru (koja postaje normala u 2D ekranskom prostoru za $w = 1$, što odgovara projiciranju na danu ravninu tj. sam framebuffer) preko vektorskog produkta radijvektora v_1 i v_2 (vrhovi v_1 i v_2 su uzeti u smjeru suprotnom od kazaljke na satu!):

$$v_1 \times v_2 = \begin{bmatrix} y_1 w_1 w_2 - w_1 y_2 w_2 \\ w_1 x_2 w_2 - x_1 w_1 w_2 \\ x_1 w_1 y_2 w_2 - y_1 w_1 x_2 w_2 \end{bmatrix}. \tag{23}$$

Ponavljanjem toga još dva puta za ostala dva brida (svaki put uzimajući odgovarajuće krajeve) dobivamo sva tri vektora normale bridova. Ono što ćemo moći uočiti iz općeg oblika (23) jest da se ovi vektori pojavljuju kao reci adjunkte **A** od matrice **M**, naime:

$$\mathbf{A} = \begin{bmatrix} a_0 & b_0 & c_0 \\ a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \end{bmatrix}, \tag{24}$$

gdje su

$$a_0 = y_1 w_1 w_2 - w_1 y_2 w_2 \tag{25}$$
$$a_1 = y_2 w_2 w_0 - w_2 y_0 w_0 \tag{26}$$
$$a_2 = y_0 w_0 w_1 - w_0 y_1 w_1 \tag{27}$$
$$b_0 = x_2 w_2 w_1 - w_2 x_1 w_1 \tag{28}$$
$$b_1 = x_0 w_0 w_2 - w_0 x_2 w_2 \tag{29}$$
$$b_2 = x_1 w_1 w_0 - w_1 x_0 w_0 \tag{30}$$
$$c_0 = x_1 w_1 y_2 w_2 - x_2 w_2 y_1 w_1 \tag{31}$$
$$c_1 = x_2 w_2 y_0 w_0 - x_0 w_0 y_2 w_2 \tag{32}$$
$$c_2 = x_0 w_0 y_1 w_1 - x_1 w_1 y_0 w_0. \tag{33}$$

Uspoređivanjem tri vektora koja smo dobili prije s $(a_0, b_0, c_0), (a_1, b_1, c_1)$ odnosno (a_2, b_2, c_2) recima iz **A**, vidimo da se oni u potpunosti preklapaju. Dakle, jednadžbe bridova koje ćemo positovjetiti s normalama odgovarajućih bridova, možemo implicitno dobiti iz adjunkte od matrice trokuta. Za adjunktu **A** jasno vrijedi da je ona jednaka $\det M\ \mathbf{M}^{-1}$; vidjet ćemo da možemo zanemariti utjecaj $\det \mathbf{M}$. Isto tako, vidimo da je

$$\det \mathbf{M} = c_0 w_0 + c_1 w_1 + c_2 w_2, \tag{34}$$

iz čega postaje jasno da nam je determinanta važna samo u smislu njenog predznaka. Naime, kada bi bilo $w_0 = w_1 = w_2 = 1$, tada bi predznak od $\det \mathbf{M}$ bio indikacija orijentacije trokuta: pod pretpostavkom da su svi orijentirani suprotno kazaljci na satu, ona je pozitivna kada je trokut okrenut prema kameri, a inače je negativna (to se može obrnuti u smislu da se traži da rasterizator promijeni predznak pri računanju ove formule, ako korisnik tako želi) – to je dio implicitnog obrezivanja kojeg smo spomenuli, u ovom slučaju jednostavnog odbacivanja trokutova koji su okrenuti na nama nevidljivu stranu (engl. *back-face culling*). Budući je moguće da w_i-jevi poprimaju različite vrijednosti od 1, raspisom (34) dobivamo izraz iz kojeg je moguće izlučiti $w_0 w_1 w_2$, iz čega slijedi da, ako su svi w_i pozitivni, determinanta ostaje ista; ako su svi negativni, determinanta mijenja predznak i to znači da se cijeli trokut nalazio iza kamere (to odgovara negativnom z-u, budući je općenito $w = z$ do na neki konstantni pozitivni faktor) pa taj slučaj zanemrajujemo (oni se tipično rješavaju u prethodnom stadiju renderiranja); problem može biti samo u slučaju da su jedan ili dva w_i-ja negativna te taj slučaj odgovara situaciji kada je jedan odnosno dva vrha trokuta iza kamere, a ostatak ispred; to je situacija koja se isto tako ne bi smijela dogoditi u ovom stadiju, budući da to implicira da trokut siječe $z - near$ ravninu koja ga je prije trebala obrezati i ovdje dostaviti samo trokute između te dvije ravnine. Time smo dakle dokazali da su za pod našim pretpostavkama, predznaci tih dviju determinanti jednaki i možemo koristiti predznak od (34) za brzo odbacivanje trokutova *prije* nego što ih uopće počnemo rasterizirati, dakle tipično u fazi postavljanja.

Sada kada znamo jednadžbe bridova (normale) $E_i(x, y)$ – one su redom reci $i \in \{0, 1, 2\}$ iz **A** – možemo izračunati skalarni produkt piksela $(x, y, 1)$ i dane normale (a, b, c); pošto su normale već prirodno retčani vektori iz adjunkte, to možemo pisati kao

$$E(x, y) = [a, b, c] \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{35}$$

Ovaj račun je vrlo brz i omogućava inkrementalnu evaluaciju preko činjenice da je $E(x+s,y+t) = E(x,y)+as+bt$, što je važno radi što brže evaluacije atributa pri šetnji kroz Hilbertovu krivulju.

Analogon zanemarivanja krajnjih desnih i krajnjih donjih piksela u prošlom tipu rasterizatora radi osiguravanja svojstva 3.3 ovdje predstavljaju određeni dopušteni slučajevi predznaka izračunatih triju skalarnih produkata s bridovima – kada su sva tri pozitivna, sigurno smo u trokutu. Naime, budući da susjedni trokutovi imaju suprotne normale, može se dogoditi da pri testiraju predznaka za neke od bridova dobijemo 0, pa tada koristimo daljnje testove za odlučivanje za taj brid, da se ne dogodi dvostruko iscrtavanje piksela: ako želimo ekvivalent našoj prošlo odabranoj gornje-lijevoj konvenciji, moramo (u slučaju da dobijemo 0) testirati je li $a > 0$ (normala lijevog brida ima pozitivnu x komponentu); ako ispadne da je i on 0, tada moramo ići još jedan korak dalje i testirati je li $b < 0$ (normala gornjeg brida ima negativnu y komponentu) – ovdje više ne možemo dobiti da je $b = 0$ jer bi to značilo da je normala nulvektor, što nije moguće u nedegeneriranim trokutovima (takvi bi trebali biti izbačeni u fazi procesiranja vrhova tj. u hipotetskom procesoru vrhova.).

3.4.2 Evaluacija atributa u trokutu preko jednadžbi bridova

Osim testiranja pripadnosti trokutu, nužno je znati i trenutne vrijednosti svih atributa koji se interpoliraju uzduž trokuta na trenutnoj koordinati piksela; to je nešto što funkcionira malo drukčije u ovom tipu rasterizatora od do sada obrađenih postupaka.

Iz (6) znamo kako izračunati vrijednost danog atributa na koordinati određenoj baricentričnim koordinatama koje pak dobijemo iz Cramerovog sustava

$$(x,y,1) = \alpha(x_0w_0, y_0w_0, w_0) + \beta(x_1w_1, y_1w_1, w_1) + \gamma(x_2w_2, y_2w_2, w_2). \tag{36}$$

Rješenje tog sustava su redom determinante

$$\alpha = \frac{\begin{vmatrix} x & x_1w_1 & x_2w_2 \\ y & y_1w_1 & y_2w_2 \\ 1 & w_1 & w_2 \end{vmatrix}}{D} \tag{37}$$

$$\beta = \frac{\begin{vmatrix} x_0w_0 & x & x_2w_2 \\ y_0w_0 & y & y_2w_2 \\ w_0 & 1 & w_2 \end{vmatrix}}{D} \tag{38}$$

$$\gamma = \frac{\begin{vmatrix} x_0w_0 & x_1w_1 & x \\ y_0w_0 & y_1w_1 & y \\ w_0 & w_1 & 1 \end{vmatrix}}{D}, \text{gdje je} \tag{39}$$

$$D = \begin{vmatrix} x_0w_0 & x_1w_1 & x_2w_2 \\ y_0w_0 & y_1w_1 & y_2w_2 \\ w_0 & w_1 & w_2 \end{vmatrix}. \tag{40}$$

Uočavamo da je D jednako (34).

Međutim, u ovom rasterizatoru radimo s homogenim koordinatama te ne vrijedi nužno da je $w_i = 1$, stoga ako prvo podijelimo sve vrhove s odgovarajućim w_i (što predstavlja ekvivalentnu točku u projektivnoj ravnini) i zatim u (36) uzmemo $(x_i,y_i) = (x_i,y_i,1)$, dobivamo sumu čija je homogena koordinata jednaka $\alpha+\beta+\gamma$ – ta trojka se upravo dobije iz (37). To znači da je ekvivalentna normalizirana točka

$$(x,y,1) = \frac{\alpha(x_0,y_0,1)+\beta(x_1,y_1,1)+\gamma(x_2,y_2,1)}{\alpha(x,y)+\beta(x,y)+\gamma(x,y)}. \tag{41}$$

Sada primjenom funkcije $u(\mathbf{x})$ za neki atribut u (na isti način kao u prošlom rasterizatoru; naime, tu imamo jednu afinu kombinaciju pa u smijemo provesti kroz izraz) dobivamo

$$u(x,y) = u_0 f_0(x,y) + u_1 f_1(x,y) + u_2 f_2(x,y), \tag{42}$$

gdje su

$$f_0(x,y) = r\alpha(x,y) \tag{43}$$
$$f_1(x,y) = r\beta(x,y) \tag{44}$$
$$f_2(x,y) = r\gamma(x,y) \tag{45}$$
$$r = \frac{1}{\alpha(x,y) + \beta(x,y) + \gamma(x,y)}. \tag{46}$$

Iz forme dobivenih determinanti iz (37) uočavamo da su te determinante točno jednake skalarnim produktima $(x,y,1)$ s odgovarajućim kofaktorima korištenim u adjunkti (24) nakon što se $\det \mathbf{M} = D$ pokrati iz svakog pribrojnika u brojniku s istim u nazivniku (zato smo bili rekli da nećemo gledati determinantu u (34) – ona na kraju uopće ne utječe na evaluaciju atributa), konkretno:

$$\alpha(x,y) = E_0(x,y) \tag{47}$$
$$\beta(x,y) = E_1(x,y) \tag{48}$$
$$\gamma(x,y) = E_2(x,y), \tag{49}$$

što omogućava da iskoristimo već jedanput izračunate vrijednosti bez dodatnog računa, barem teoretski. Praktički to nije moguće jer su zahtjevi za preciznost i dinamički raspon od E_i-ova mnogo manji od onih za α, β, γ koji predstavljaju vrijednosti koje treba evaluirati na svakom pikselu trokuta kako bi se na temelju njih izračunao izraz (42) za konkretan atribut u. Atributi mogu biti najrazličitije prirode i zato ne bi bilo pametno pohranjivati oba tipa broja s istom preciznosti u 16.16 formatu s fiksnim zarezom. Zaključujemo da za razliku od prošlih pristupa, ovdje ne interpoliramo direktno svaki atribut zasebno između točaka trokuta, već jednostavno evaluiramo na svakom pikselu (po mogućnosti inkrementalno) skup triju baznih jednadžbi koje su ekvivalentne jednadžbama bridova i omogućuju perspektivnokorektnu evaluaciju svih atributa preko (42).

3.4.3 Pretvorba podataka u brojeve s fiksnim zarezom i implicitno obrezivanje

Slično kao u prošlom tipu rasterizatora, i ovdje koristimo isključivo aritmetiku s fiksnim zarezom 16.16 za evaluaciju atributa i pomicanje jednadžbi bridova kako se krećemo po Hilbertovoj krivulji. Razlog ovoga puta nisu samo peroformanse i jednostavnost hardverske implementacije već i potreba da možemo vratiti interpolirane jednadžbe točno unatrag iz najnižih nivoa Hilbertove krivulje natrag na više kako bi se šetnja nastavila od zadnje napuštene pozicije – time ustvari iterativno izvodimo rekurzivni spust, uz jedan vrlo mali fiksni stog (koji je varijabla, ne sistemski stog). No prvo moramo pretvoriti sve ulazne podatke – jednadžbe bridova i jednadžbe interpolacije atributa – iz `float` tipa u 16.16 format za kasniji račun.

Ideja pretvorbe u format fiksnog zareza je drukčija nego u prošlom tipu rasterizatora, gdje nismo mogli izbjeći činjenicu da veliki dinamički rasponi u atributima mogu dovesti do nereprezentabilnih 16.16 vrijednosti. Ovdje pak imamo E_i i α, β, γ jednadžbe koje predstavljaju iste jednadžbe bridova trokuta, ali se moraju drukčije tretirati na numeričkoj razini budući da se upotrebljavaju na sasvim različite načine: to nam omogućava da prilagodimo način pohrane odgovarajućih vrijednosti i izbjegnemo velike greške kod evaluacije atributa koja je pak sasvim nezavisna od preciznosti testiranja pripadnosti trokutu (za koju su važni *samo* E_i-ovi).

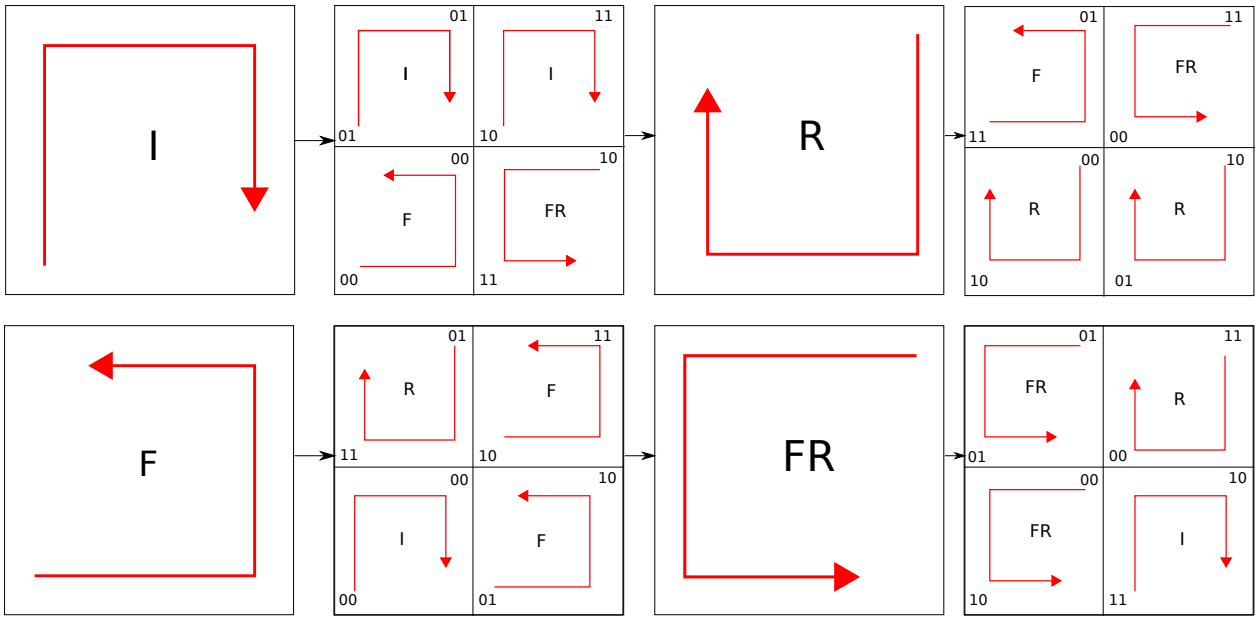
Kod E_i-ova je bitna jedino reprezentabilnost različitih smjerova tj. orijentacijska preciznost koja ima veze samo s a i b; c pak predstavlja udaljenost brida od ishodišta u jedinicama duljine normale i može

se slobodno skalirati, dok god dakako i a i b bivaju skalirani istim faktorom (ovo su sve homogene jednadžbe hiperraravnine – možemo ih množiti sa svime osim 0 i dalje će predstavljati isti brid). To znači da bi bilo najbolje odabrati najvećeg po apsolutnoj vrijednosti od a i b jer tada znamo koliki je raspon vrijednosti ove jednadžbe i možemo podijeliti sve faktore s tom maksimalnom vrijednosti kako bismo osigurali da je cjelobrojni dio što manji i stoga cijela vrijednost reprezentabilna u 16.16 varijabli. Točnije, to je u ovom radu implementirano na način da se odabere faktor s najvećim eksponentom k (u smislu IEEE 754 formata `float`) i zatim pomaknemo (desni shift) mantise od a, b i c za k, što a i b dovodi u raspon $[-1,1]$, s tim da c može postati premali da bi bio reprenzentabilan u 16.16 formatu pa ga u tom slučaju zaustavimo na (po apsolutnoj vrijednosti) minimalnoj 16.16 vrijednosti. Duljina normale (a,b), R, je sada očito u rasponu $[1,\sqrt{2}]$, pa vrijednost c-a predstavlja udaljenost brida od ishodišta pomnoženu s R. Pretpostavimo da je najdulja stranica framebuffera duljine V, tada je njegova dijagonala duljine $V\sqrt{2}$, a budući da je $R \in [1,\sqrt{2}]$ vrijedi da je $|c| \leq 2V$ ukoliko se brid nalazi unutar ekrana tj. ukoliko mu je udaljenost od ishodišta najviše $V\sqrt{2}$. Time smo upravo dobili vrlo jednostavan dovoljan (ali ne i nužan) način za obavljanje uklanjanja trokutova van ekrana: ako je c trenutnog brida trokuta izvan tog raspona, brid je definitivno van ekrana – pitanje je samo vrijedi li to za cijeli trokut, što možemo otkriti testiranjem u kojem poluprostoru (koje separira jednadžba brida) leži ishodište – ako je ono u negativnom poluprostoru (onom u kojem je test skalarnog produkta negativan), tada je trokut sigurno izvan ekrana; inače, trokut nije nužno u cjelosti izvan ekrana, ali se može izbaciti trenutni *brid* iz rasterizacije jer nikada neće biti potrebno testirati se s njim u rasterizaciji. Ova provjera pripadnosti ishodišta se ustvari svodi na skalarni produkt $(0,0,1)$ s (a,b,c), što onda očito ovisi samo o predznaku c-a. Primijetimo da ovdje možemo *nezavisno* birati maksimalne eksponente od a i b za sva tri brida pošto se testiranje piksela provodi sa svakim bridom nezavisno.

S druge strane, pretvorba α, β i γ jednadžbi ne može koristiti nezavisnost koja vrijedi za E_i-jeve; iako se radi o fundamentalno istim jednadžbama, bazne jednadžbe atributa se u (42) *množe* s vršnim vrijednostima atributa i zatim zbrajaju, dok se E_i-jevi samo koriste za predznak jednog skalarnog produkta. To znači da je ove bazne jednadžbe potrebno smatrati *sve zajedno*, u cjelini – ne možemo na različite načine u 16.16 format pretvarati različite vrijednosti! Pristup koji pokazuje dobre rezultate za pretvorbu ovih vrijednosti je uzimanje vrijednosti koeficijenta u_i s najvećim eksponentom od svih ostalih koeficijenata svih ostalih atributa; nakon toga, smanjimo (skaliranjem) sve koeficijente svih atributa sve dok najveći koeficijent uđe u svoj reprezentabilan raspon – ovaj postupak garantira maksimalni dinamički raspon atributa i preciznost interpolacije. Potom, pri evaluaciji i samom iscrtavanju, vraćamo se u originalnu skalu množenjem rezultata dobivenog iz (42) s potonjim kvocijentom i iskoristimo dobiveni broj kao konačnu vrijednost atributa na danom pikselu[12].

Spomenimo još i mogućnost obrezivanja piksela koji su van ekrana; naime, način evaluacije predznaka skalarnih produkata koji je u centru ovog pristupa se može vrlo dobro iskoristiti i za ovu svrhu. Ovakvo obrezivanje je različito od gore spomenutih koraka u kojima se može odbaciti određene trokute – ovdje se zaista garantira da bilo koji piksel koji je izašao van ekrana (jer je takav trokut) neće biti iscrtan tj. imat će negativan rezultat testa pripadnosti. To se očito može postići dodavanjem svakom trokutu još četiri jednadžbe (koje je potrebno testirati uz ostale tri jednadžbe brida (u istom formatu normale $[a,b,c]$, kao za bridove trokuta) za svaki potencijalni piksel) koje opisuju naš pravokutni ekran s ishodištem $(0,0)$ u

[12]Tu očito postoje i drugi mogući pristupi – jedan od njih je npr. pomnožiti r iz (43) s tim faktorom potrebnim za normalizaciju najvećeg koeficijenta u (42). Na taj način se taj faktor krati pri interpolaciji te na kraju dobivamo ispravnu vrijednost, no problem je u tome što na ovaj način r može lako preći reprezentabilan raspon u 16.16 zbog potencijalno velikih faktora korekcije, stoga je primarno predložena metoda bolja – na kraju ionako dobivamo broj koji je u najgorim slučajevima van 16.16 raspona, no tada ga možemo evaluirati i izračunati kao `float` te koristiti takav broj za uzorkovanje/daljnje piksel procesiranje.



Slika 3. Hilbertova krivulja definirana putem fraktalnih limesa (koji staju na najnižoj razini pojedinačnih piksela)[10].

donjem lijevom kutu, dimenzija (w,h), počevši od one lijevog brida:

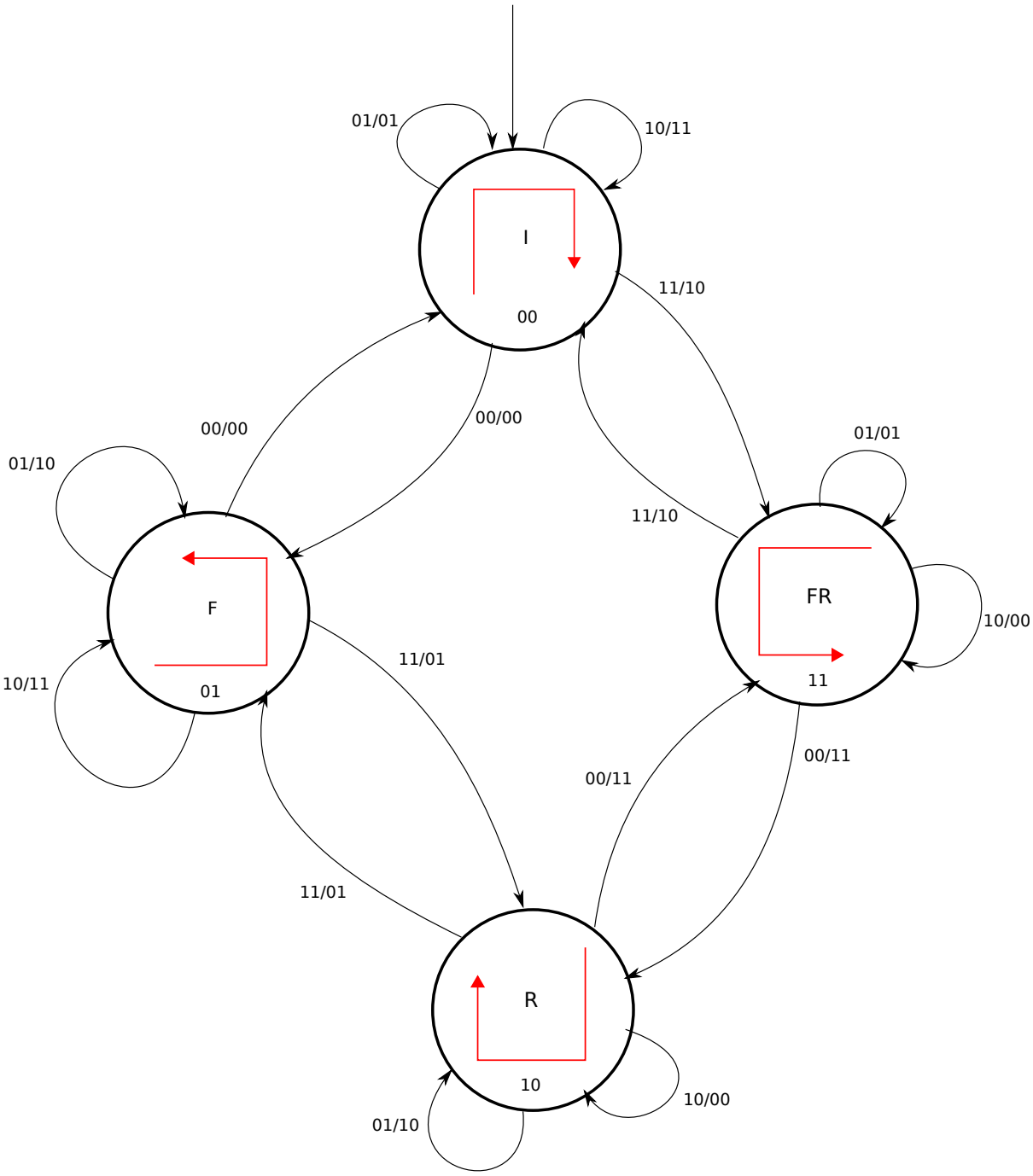
$$d_0 = [1,0,0] \tag{50}$$
$$d_1 = [-1,0,w] \tag{51}$$
$$d_2 = [0,1,0] \tag{52}$$
$$d_3 = [0,-1,h]. \tag{53}$$

3.4.4 Glavni algoritam

U središtu ovog algoritma je Hilbertova krivulja koja se može definirati kao limes skupa formalnih pravila prikazanih u 3.

Naravno, naša je slika konačna i stoga koristimo aproksimaciju Hilbertove krivulje koja se tipično zove aproksimacija reda N – ona aproksimacija koja se dobije primjenom u nizu samo prvih N pravila. Najlakše ćemo intuitivno shvatiti ovu krivulju ako razmišljamo o slici kao prezentaciji različitih načina za obići četiri različite rotacije i/ili refleksije istog osnovnog oblika: „čaše" tj. kvadrata $2^N \times 2^N$ s otvorenim jednim vrhom (bridom). Npr. `I` predstavlja oblik kod kojeg počinjemo dolje lijevo i moramo doći do donjeg desnog kraja – to činimo u četiri koraka koja su sama po sebi opet jedan od četiri već spomenuta oblika – svaki od ovih oblika tako ima jednoznačan način za doći do kraja koristeći različite kombinacije osnovna 4 oblika. Ovaj postupak se očito rekurzivno nastavlja za svaku razinu, pri čemu svaki put kada nacrtamo dio puta do cilja moramo zamisliti da smo ušli u taj podoblik i u njemu rekurzivno nastavili isto iscrtavanje, samo ovoga puta na $1/2$ početnih dimenzija (četvertini površine). Sada postaje jasno kako ovaj algoritam pomaže ovoj rasterizaciji – postpunim dijeljenjem ekrana na ovakve sve manje dijelove omogućavamo brzo hijerarhijsko testiranje i rano prekidanje dodatne rekurzije ukoliko ispadne da u velikoj regiji nema trokuta, ali i bolje iskorištenje cachea jer nam Hilbertova krivulja garantira da ćemo se uvijek nastaviti kretati u neposrednoj blizini početnih piksela, prateći susjede što je više moguće dok tako ne dođemo do nivoa piksela te se zatim vratimo na prošli nivo gdje opet nastavljamo u pravilnom nizu koraka iz definicije krivulje. Također vidimo zašto je potrebno početi s kvadratnim $2^N \times 2^N$ framebufferom: svako novo silaženje u Hilbertovoj krivulji zahtijeva prepolovljavanje prošlih dimenzija, a pošto mi želimo doći do cjelobrojne najniže razine piksela, svi gornji nivoi moraju biti dijeljivi s 2.

Gornja slika daje funkcionalan prikaz Hilbertove krivulje u obliku konačnog automata stanja koji ima



Slika 4. Hilbertova krivulja se može definirati konačnim automatom, ovdje prikazan kao Mealyev automat [10].

četiri stanja iz već spomenutih razloga. Unutar čvorova (krugova) piše dvobitni identifikator stanja; na usmjerenim bridovima (strelicama) stoji ij/xy, gdje bitovi ij predstavljaju dužinu krivulje na trenutnoj razini – to možemo najlakše vizualizirati kao brojač koraka koje smo napravili na trenutnoj razini od početne pozicije iz prošle slike do krajnje pozicije – na početku, on je 0, a na kraju 3. xy pak predstavlja odgovarajuću x odnosno y koordinatni, po jedan bit svaka. Dakako, taj jedan bit služi samo za ovu trenutnu razinu – kako se krećemo po različitim razinama, tako se i bitovi zapisuju na druge (niže) bit-pozicije. Možemo zamisliti da je na prvoj, početnoj razini krivulje na početnoj lokaciji duljina 00 i lokacija 00; na kraju (dužina 11) je pak pozicija 10 jer smo se pomakli za 2^N piksela (što je jednako 1 postavljenom na najvišem bitu x pozicije), horizontalno, ali se nismo pomakli vertikalno po prvom `I` pravilu! Primijetimo također da je ovaj FSM reverzibilan u smislu da kada napravimo tranziciju iz jednog stanja u drugo, uvijek se znamo vratiti natrag, što uz spomenuta svojstva reverzibilnosti plus i minus operacija s fiksnim zarezom omogućuje vrlo efikasnu implementaciju ovakvog rasterizatora bez praktički ikakvog pravog stoga. Kada se FSM sa slike pusti da radi do razine piksela i ispunjenja ekrana, dobivamo specifičnu konačnu strukturu poznatu kao *quadtree* odnosno stablo čiji su svi čvorovi izlaznog stupnja 4, osim listova na kojima su krajnji pikseli[13].

Očito, hijerarhijski pristup bi trebao lako omogućiti sprječavanje nepotrebne dodatne rekurzije, što je tu riješeno (slično kao prije kad smo eliminirali trokute van ekrana) jednostavnim testiranjem četiriju krajeva dane regije na predznak skalarnog produkta te točke s normala bridova trokutova; za to testiranje se pomiču same *jednadžbe* odnosno E_i-evi u onoj komponenti normale koja odgovara smjeru kojim treba ići za doći do odgovarajućeg ruba ekrana (to nam je najjednostavnije učiniti, budući da bi alternativa bila znati točne cjelobrojne (x,y) koordinate piksela, no one kao takve nisu poznate – poznat je samo 1 bit po koordinati za trenutnu razinu krivulje, sve ostale traži više pohranjivanja!). Na kraju, zaključak donosimo prema tome jesu li *svi* krajevi bili negativni na barem jednom testu – to tada očito znači da u ovoj regiji nema trokuta i daljnja rekurzivna subdivizija može biti spriječena. Inače, nastavljamo dijeliti regije na gore opisan način prateći FSM. Kako se pomičemo po lokacijama pojedinih stanja, tako se i jednadžbe E_i i f_i ažuriraju u toku, na sličan način kao kod testiranja pripadnosti trokuta u regiji. To znači da c-ovi jednadžbi bridova uvijek predstavljaju udaljenost od trenutnog donjeg lijevog piksela *regije*, jer smo počeli u donjem lijevom pikselu framebuffera što se onda kroz rekurziju u niže razine pretvorilo u manje regije. Primijetimo isto da su stanja koja ne počinju u donjem lijevom 00 kutu, stanja `FR` i `R`, koja lako prepoznamo po tome što imaju postavljen viši bit. Za takva je stanja pri silaženju niz razine krivulje potrebno uvećati xy smjesta za 11 čim dođemo tamo i onda nastaviti normalnim tijekom. Taj tijek je u potpunosti opisan FSM-om na slici i iz njega možemo izgraditi dvije glavne 32-bitne tablice: ona koja govori u koje sljedeće stanje prijeći iz trenutnog stanja s na temelju trenutne duljine krivulje r te ona koja govori u koje se stanje vratiti na temelju trenutnog stanja s i lokacije xy. Obje tablice očito preslikavaju u 2 bita stanja iz $2\times(2\times 2)$ kombinacija stanja i duljina/lokacija – to čini 32 bita. Za kraj još postoji jedna dodatna tablica koja nam kazuje u kojem sljedećem smjeru ići nakon što se vratimo iz rekurzije u trenutni nivo te je nužna kako bi se izbjegao klasičan stog; pritom je važno vratiti sve vrijednosti prethodno ažuriranih jednadžbi na one vrijednosti koje su imale prije silaska u tu razinu, jer se sada vraćamo iz tog dijela rekurzije natrag; tu se opet koristi praktički jednak račun kao prije. Ova posljednja tablica očito ovisi o trenutnom stanju i specificira za svaku od 4 lokacije jedan smjer od 4 moguća do sljedeće lokacije (trenutnog stanja), pa i ona nosi 32 bita. Dakle, FSM možemo zakodirati u sveukupno 12 bajtova što je korisno za čisto hardverske implementacije. Konkretne tablice možete pronaći u [10].

Povrh toga, potrebno je imati i varijablu stoga fiksne veličine od $2\times N$ bitova, gdje je N broj nivoa krivulje, u našem slučaju $N=16$ pa su dovoljna 32 bita. Ta varijabla r pohranjuje dva bita za svaku razinu koja označavaju zadnju kompletno procesiranu lokaciju od njih 4, što je važno kako bi se mogli vratiti na tu razinu nakon okončanja daljnje rekurzije u niže nivoe.

Za kraj spomenimo glavnu petlju algoritma koja iterativno implementira rekurzivni spust: počinjemo u

[13] Doduše, zbog optimizacije poradi koje ne moramo silaziti u regije u kojima nema novih trokutova, ovaj *quadtree* može ispasti degeneriran.

stanju `F` jer nakon ulaska u `I` uvijek završavamo prvim spustom u `F`. Lokacija je 00 jer viši bit stanja nije još bio postavljen, te su sve jednadžbe originalne. Sada se započinje testiranje pripadnosti, spuštanje ako je potrebno, te na kraju, ako nije bilo spuštanja ili ako smo došli do piksela, poziv funkcije ažuriranja koja ostaje na trenutnoj razini ali prelazi na sljedeću lokaciju po redu, dok ne prođe sve 4. Tada se `bool` varijablom signalizira glavnu petlju da je (barem dijelom) rekurzija gotova, te ona tada nađe prethodno stanje preko tablice u koje se može vratiti kako bi se rekurzija tamo nastavila. Na taj način se sve ponavlja dok ne dođemo opet do vršne razine N, nakon čega petlja stane jer više nema prethodnih stanja – ovo je kraj. U suprotnom smjeru, kad dođemo do dna, do pojedinačnog piksela, moramo testirati na klasičan način da li je on unutar (barem nekog od) trokuta – to se ovdje radi uz jednostavnu provjeru c-a jer moramo imati na umu da se sve koordinate, pa i xy dvobitne lokacije unutar pojedinih razina krivulje, mjere u odnosu na donji lijevi rub (to je tako bilo na N-toj razini i tako rekurivno nastavlja vrijediti) – dovoljno je provjeriti predznak c-a jer očito vrijedi $E(0,0) = c$. Renderiranje piksela (pretpostavljamo da radimo samo mapiranje tekstura) zahtijeva samo relativno jednostavnu evaluaciju (42) i uzorkovanje teksela (uz neizbježno dubinsko testiranje, no tu ništa nije novo).

Za kraj, recimo samo to da se ovaj algoritam pokazao vrlo efikasnim za ekonomične hardverske implementacije koje omogućavaju kontrolu na mnogo sitnijoj razini paralelizma u izvođenju mnogih operacije koje mi u softverskoj implementaciji ne možemo postići, što samo ističe koliko je važno imati dedicirani hardver za specijalizirane zadatke poput rasterizacije, ali i ostale stadije grafičkog cjevovoda. To je moguće zahvaljujući hijerarhijskom (dijelimo ekran na sve manje podregije koje dalje rekurzivno dijelimo do piksela; odustjemo od rada što je prije moguće), inkrementalnom (pri šetnji i spustu Hilbertovom krivuljom inkrementalno ažuriramo i jednadžbe bridova i jednadžbe interpolacije atributa, koje kasnije možemo egzaktno vratiti natrag na vrijednosti iz prošle razine kako bi iterativno implementirali rekurziju, zahvaljujući svojstvima 16.16 aritmetike) pristupu, te konačno općenitom načinu rasterizacije koji se ne bazira na interpoliranju niz bridove trokuta kao 3.3, već koristi jednadžbe bridova trokuta za efikasnu provjeru pripadnosti piksela trokuta i za interpoliranje svih atributa (koje tu može biti isključivo perspektivno korektno). Prvenstvena prednost ovog pristupa je što šeće po pikselima u prostorno koherentnom redoslijedu Hilbertove krivulje, i to ne samo na najvišoj razini, nego i na svim nižim razinama, sve do samih piksela. To znači da algoritmi poput mapiranja tekstura sada mogu puno efikasnije koristiti teksturni cache. Također, on garantira da će se maksimalno iskoristiti određeni blok framebuffer cachea, jer da bismo prešli u iduću lokaciju (podregiju) stanja u Hilbertovoj krivulji, moramo u potpunosti završiti s procesiranjem prošle regije i više se nikada nećemo u nju vratiti (pod pretpostavkom da naše podijeljene regije barem na nekoj razini odgovaraju bloku framebuffer cachea, što zbog fraktalnosti podjele mora negdje vrijediti). Autori rada [10] također navode da bi paralelizacija rasterizacije više poligona mogla pomoći u performansama u slučaju kada je većina poligona jako male površine, otprilike 1 ekranski piksel – tada očito nema puno koristi od hijerarhijskih podjela i ranih odbacivanja, no ovaj algoritam i dalje ima ostala dobra svojstva, pogotovo u paralelnoj izvedbi koja se koristi u našoj implementaciji. Ona za razliku od 3.3 ne pati od problema sinkronizacije rada različitih niti – svaka nit ovdje dobiva svoju potpuno nezavisnu kvadratnu regiju framebuffera i nad njime izvodi rasterizaciju[14] i pritom pristupa trokutima.

Prije nego što pomislimo da su ovakve vrste optimizacija u dalekoj prošlosti te da rasterizatori danas „jednostavno rade", spomenimo da je AMD u svoju novu Polaris arhitekturu 2016. godine ugradio mnogobrojne optimizacije vezane upravo za rasterizaciju i posljednje korake pretprocesiranja prije same

[14] Ključan korak u našoj implementaciji je inicijalna podjela početnog „proširenog" framebuffera na n kvadratnih regija $2^m \times 2^m$, otprilike jednake površine; pritom se mora također izbaciti sve trokutove koji uopće ne pripadaju toj regiji, za što se primjenjuje u osnovi identičan postupak onom prije opisanom pri spuštanju Hilbertovom krivuljom, samo što se on ovdje primjenjuje na početnom, najvišem nivou pojedine Hilbertove podkrivulje. To je potrebno zato što mi pri rekurzivnom spustu mijenjamo pohranjene jednadžbe bridova – kad bismo to radili nezavisno sa svake niti, došlo bi do konflikta u toj kritičnoj sekciji. Isto je potrebno raditi i zbog jednadžbi interpolacije atributa, ali ne i za same atribute koji ostaju nepromjenjeni tijekom cijele rasterizacije. Primijetimo da je moguće da neki trokuti sijeku rubove podregije, pa ih onda moramo kopirati u lokalne memorije svih susjednih podregija tj. svih niti kojima su one dodijeljene. To je samo jedan moguć pristup paralelizaciji; za pristupe (više ih je) koji su korišteni u ovom radu, pročitajte izvorni kôd i popratne komentare.

rasterizacije, uključujući i podjelu ekrana na regije unutar kojih rasterizacijske jedinice mogu renderirati jedan trokut u jednom ciklusu (slično vrijedi i za ovdje opisan algoritam kad se implementira na usko specijaliziranom hardveru s mnogo šansi mikroparalelizma na razinama koje su nedostupne u softveru[10]), uklanjanje degeneriranih trokutova te filtriranje onih trokutova koji su toliko mali da ne doprinose niti jednom konačnom pikselu. Sve to zajedno, uz dodatne inovativne trikove poput *lossless* kompresije zasebnih regija framebuffera na lokalno optimalan način (relativno jednostavna delta kompresija), dovodi do 3x ubrzanja u pojedinim scenama te 40% uštođene električne energije u usporedbi s prošlim arhitekturama[11]. Sve u svemu, možemo zaključiti da su ovakva istraživanja i algoritmi i dalje još vrlo važni te se ne čini da će ikada prestati biti vruća tema u razvoju novog hardvera, ali i softvera nad njime sa strane programera koji žele izvući maksimalne performanse uz sve složenije efekte i scene i to ne samo u igrama već i u ozbiljnim CAD, CAM i sličnim aplikacijama (Blender, Darktable itd.).

3.5 Završne primjedbe o rasterizaciji

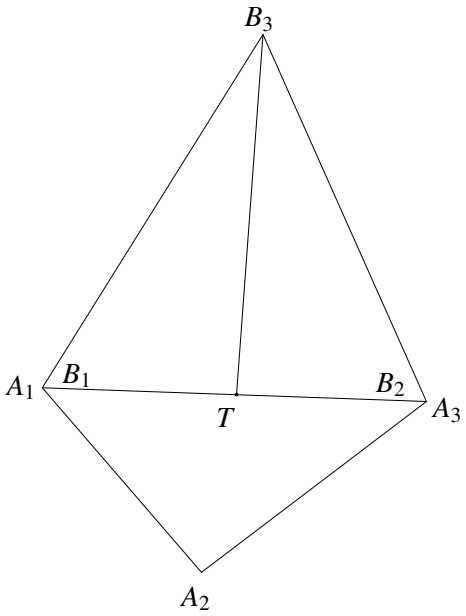
Konačno, spomenimo neke probleme koji se mogu pojaviti pri rasterizaciji i predložimo moguća rješenja.

3.5.1 Greške zbog T-raskrižja

Jedan od standardnih problema koji se pojavljuje u raznim grafičkim aplikacijama jest onaj renderiranja T-raskrižja (engl. *T-junction*). T-raskrižja su takvi dijelovi nekog modela gdje je jedan trokut susjedan s jednim drugim ili više trokutova i to preko *jednog zajedničkog* brida koji ih separira. To je odmah u kontradikciji sa pretpostavkom 3.3 i stoga se time ne bi trebali ni zamarati u rasterizatorima, no valja objasniti zašto je to problem. Naime, takvi trokutovi će se pri nepreciznoj aritmetici s pomičnim (ali isto bi vrijedilo i s fiksnim) zarezom u nekom trenutku transformirati na način da se neki vrh između krajeva prvog trokuta na separirajućem bridu transformira na numerički različitu lokaciju od one koju je trebao imati da uistinu leži na tom jednom te istom bridu; pošto prvi trokut *nije* subdiviran na način da se ti manji segmenti separirajućeg brida smatraju manjim stranicama podtrokutova, izgledat će kao da susjedni trokutovi malim dijelom lebde iznad prvog trokuta ili, još gore i očitije, da postoji jako tanka pukotina između tih susjednih trokutova. Riječ je dakle o problemu nestabilnosti koja nastaje zbog neprecizne aritmetike, no s njom se uspješno borimo time da stabiliziramo rasterizaciju na način da se uvijek kada postoji neka linija (brid) između točaka, taj isti suprotni brid, ne subdiviran, nalazi i u suprotnom poluprostoru, što znači da, ako su sekvence svih računa vrhova jednake, neće moći doći do bilo kakvih vidljivih grešaka. Ovaj se proces najčešće obavlja automatski kao dio geometrijskih kompajlera ili optimizatora i moguće ga je implementirati kao zadatak nad dualnim grafom početnog povezanog planarnog grafa mreže (s trokutastim licima).

3.5.2 Greške zbog računa s pomičnim zarezom

Do sada smo se bavili samo specifičnostima aritmetike s fiksnim zarezom, no ima i nekoliko stvari na koje valja pripaziti pri radu s pomičnim zarezom, koji danas apsolutno dominira u svim računskim operacijama koje izvodi procesor radi transformacije geometrije tj. račun potrebnih matrica koje se tada predaju GPU-u. Naime, aritmetika s pomičnim zarezom nije asocijativna niti komutativna, što znači da ako želimo da nam se svi vrhovi koji leže na separirajućim bridovima trokutova transformiraju *bit-identično* (što je nužno za ispunjenje 3.3), moramo osigurati izvođenje ne samo jednakih operacija s jednakim matricama nad tim vrhovima, već i *identičan redoslijed* svih FPU operacija. Usto, treba biti siguran da sve niti (ukoliko ih se koristi više za ovu pripremnu fazu renderiranja) procesora koje se bave transformacijom imaju identično konfiguriranu okolinu za rad s pomičnim zarezom. Inače, mogli bismo dobiti opet razne pukotine između trokutova gdje one apsolutne ne bi smjele biti.



Slika 5. Najjednostavniji primjer T-raskrižja s tri trokuta. Trokuti B_1TB_3 i TB_2B_3 dijele jednu zajedničku stranicu i tu nema problema; problemi nastaju zbog subdiviranja stranice A_1A_3 trokuta $A_1A_2A_3$ na dvije paralelne stranice A_1T i TA_3, za što je zaslužan označeni vrh V.[13]

Problem je u tome što je imalo drukčiji rezultat od transformacije separirajućeg brida kod jednog od trokuta dovoljan da uzrokuje skok tog brida za piksel prema van (suprotno normali tog brida), što dovodi do očite rupe ili pukotine.

Da se sve transformacije obavljaju u aritmetici s fiksnim zarezom, do ovih grešaka ne bi dolazilo jer znamo da je ta aritmetika asocijativna i komutativna[15].

4 Optimizacija raytracer-a za višenitno izvođenje

4.1 O raytracer-u

U ovom posljednjem poglavlju željeli bismo završiti našu „šetnju" kroz tehnike 3D renderiranja s objašnjenjem jednog raytracera optimiziranog za rad na modernim GPU-ovima. Naime, do sada smo se fokusirali na rasterizaciju kao još uvijek najdominantniju metodu renderiranja u svim industrijski priznatim API-jevima i hardveru, no u posljednje vrijeme su se počele javljati prve vrlo sposobne i relativno lako dostupne hardverske implementacije raytracing API-ja u nekim *high-end* GPU-evima.

Raytracing predstavlja fundamentalno različit pristup generiranju konačne slike na ekranu koji može lakše rezultirati mnogo realističnijom slikom s mnogo zanimljivih i precizno simuliranih optičkih fenomena koje je gotovo nemoguće postići rasterizacijom. Ipak, općenitost ovakvog pristupa i rezultantna moć dolaze po visokoj cijeni, budući da su naivne implementacije raytracer-a mnogo sporije od rasterizatora. Osnove raytracing-a čitatelj može pronaći u prošlom radu [1], no ovdje će biti dovoljno samo pročitati neke osnovne metode u popratnom izvornom kodu kako bi se shvatilo o čemu se radi (pod pretpostavkom da se shvatilo sve do sada iskazano u ovom radu).

Naime, implementacija raytracer-a koja je integralni dio ovog rada je nastala kao plod višenitne optimizacije originalnog osnovnog raytracer-a koji je bio namijenjen za jednostavno izvođenje na jednoj niti/jezgri CPU-a. To je bio običan uvodni raytracer kojim se pokazuju osnove 3D renderiranja, bez

[15] Za ovime nema potrebe i bilo bi štetno po performanse modernih (mikro)arhitektura CPU-ova zbog toga što bi se preoptrećivalo ALU jedinice dok FPU praktički uopće ne bi bio korišten. To je samo jedan od najočitijih razloga zašto ovo treba izbjegavati i umjesto toga slobodno koristiti brojeve s pomičnim zarezom, ali s gore navedenim oprezom.

posebnih optimizacija. Nije imao nikakve funkcionalnosti renderiranja složenih 3D mesheva, već je jednostavno radio s eksplicitno deklariranim oblicima u kodu.

S dolaskom AMD-ovog ROCm-a i novijih GPU-ova iste tvrtke, postalo je moguće na vrlo niskoj razini programirati ove vrlo sposobne GPU-ove i na taj način postići puno veću razinu paralelizma od one koju možemo očekivati na današnjim CPU-ovima. To je bio primarni motivator za pokušati portati originalni kod tog jednostavnog raytracer-a na ROCm platformu koristeći AMD-ovu HIP ekstenziju jezika C++. Pritom, raytracer je poboljšan dodavanjem određenih funkcionalnosti, poput učitavanja OBJ mesheva, informacija o teksturama/materijalima te dodavanjem koda za teksturiranje i Phong sjenčanje. Sve ovo je implementirano od nule i jako malo originalnog koda je na kraju ostalo kao dio novog projekta; retrospektivno možemo reći da bi bilo bolje ovakav raytracer implementirati bez korištenja ikakvih objektno-orijentiranih konstrukata, jednostavno zato što HIP u trenutku pisanja nije podržavao virtualne metode koje su praktički jedini razlog (točnije, polimorfizam) zašto bi netko i htio razmišljati o ovom problemu na objektno-orijentirani način. U nedostatku te fundamentalne sposobnosti, većina „prednosti" objektno orijentirane metodologije pada u vodu te nam ostaje samo masa neproduktivnog teksta koji tek u nekoj kasnijoj fazi čitanja i razmišljanja može postati kod. Iako je raytracing (i razne vrste simulacija općenito) školski primjer primjene objektno orijentiranog programiranja, uvjerili smo se na ovom projektu da je to daleko od istine u praksi, što potvrđuju i ostali praktičari u svom radu.

Nadalje, ovaj novi raytracer podržava i čisto CPU renderiranje kako bi se mogao raditi benchmarking GPU-a na efektivan način. Isti build može renderirati na oba načina uz korištenje odgovarajuće opcije u terminalu; detalji su u popratnoj dokumentaciji koja je dio izvornog koda. Napomenimo da je i dalje nužno imati AMD GPU i `hipcc` kompajler kako bi se uopće mogao buildati ovaj projekt, budući da je cijeli smisao upravo (na kraju) izvođenje na GPU-u. Za uspješan build trenutno podržavamo samo hipcc 2.10 (detalji su u README-u) i Ubuntu 18 LTS, budući da je autor radio na takvom stroju kojeg AMD službeno podržava te s tom hipcc verzijom (poznato je da se događa određena regresija u programu s novijim hipcc verzijama tako da je za sada nužno koristiti ovaj stariji point release; jedna moguća alternativa za novije HIP verzije je predstavljena u zasebnom direktoriju).

4.2 Tehnike korištene u optimizaciji

Pri pisanju koda za GPU raytracing, vodili smo se osnovnim principima koji su poznati iz polja masivno paralelnih i heterogenih sustava. Osnovni rad raytracer-a se svodi na emitiranje zraka iz točke očišta/kamere prema destinaciji na virtualnom ekranu u prostoru kamere koja odgovara pikselu prozora/ekrana na kojeg iscrtavamo sliku. Za razliku od rasterizatora, u raytracer-u se ne moraju obavljati gotovo nikakve dodatne transformacije točaka iz njihove pozicije u svjetskom prostoru za konačno renderiranje: uz samo osnovne translacije i rotacije radi animacije, dovoljno je izračunati sjecište svake zrake i odgovarajuće točke na objektu, ako takav objekt postoji na putu pojedine zrake. Očito, ovaj problem može biti vrlo zahtjevan ako se obavlja striktno sekvencijalno za svaki piksel i ako se pri svakom pikselu opet ide provjeravati isti skup potencijalnih objekata – riječ je barem o $O(nm)$ algoritmu, što ga čini praktički nemogućim za imalo veće rezolucije na standardnim procesorima.

Ovdje nam puno može pomoći visoka paralelnost GPU-ova i neke daljnje optimizacije koje će ovdje biti spomenute, ali od kojih nisu još sve implementirane u ovom raytracer-u. Naime, ako pustimo da jedan piksel odgovara jednoj zraci, tada je očito rješenje dodijeliti po jednu nit svakoj zraci, gdje će svaka nit izvoditi relativno presjecanje zraka i potencijalnih prepreka.

Ovaj algoritam je sada (za svaku nit) linearan po broju objekata u sceni, što je veliko poboljšanje otprije. Ipak, na neke stvari valja pripaziti: kernel koji napišemo bi trebao koristiti što manje lokalne memorije, kako bi mu sve varijable stale u registre Compute Unit-a koji ga izvodi bez potrebe prelijevanja u sporiju dijeljenu memoriju. Ovdje do izražaja dolazi veličina blokova i mogućnost izvođenja tzv. *memory coalescing*-a tj. podešavanja dimenzija blokova na način da se operacije čitanja/pisanja dijeljene memorije iz niti tog bloka mogu obavljati u većim blokovima nego što bi to inače bilo moguće – tako je primjerice

moguće zapisati podatke na 64 susjedne memorijske adrese ukoliko to zapisivanje gotovo istovremeno zatraži 64 niti poredanih u jednom redu (oni djeluju nad 2D poljem – slikom).

Zbog svih navedenih faktora, zaključili smo da su optimalne dimenzije blokova 64×4, s tim da po bloku može biti maksimalno 1024 niti, što nam osigurava kompatibilnosti s praktički svim modernim AMD GPU-ovima. Svi ovi brojevi su kompajlerske konstante definirane u datoteci `SlikaGPU.h`. Glavni kernel za renderiranje je implementiran u metodi `rayKernel`, koja potom poziva mnoge različite *device* metode ovisno o kojim objektima na sceni se radi.

Sve do sada rečeno se tiče samog kernela za *iscrtavanje* 3D scene, no kao što smo već rekli, potrebno je ipak (po mogućnosti) napraviti neke osnovne transformacije prije samog iscrtavanja, što bi (zbog separacije zadataka i preglednosti te lakšeg održavanja koda) trebao raditi zaseban kernel. Upravo zato smo napisali i metodu `transformKernel` koja ima upravo tu ulogu – proći kroz sve tipove objekata (trenutno: sfera, trokuta i mesh modela) te pozvati njihove transformacijske metode na GPU-u. To je izvedeno na način da svaka nit obrađuje po jednu odgovarajuću 3D „jedinicu": za sfere i trokue, to su same sfere ili trokuti, no za mesh modele su to pojedinačni njihovi trokuti, od kojih svaki dobiva po jednu nit i tako se može vrlo brzo procesirati relativni složen model, opet zahvaljujući visokom paralelizmu.

Zadnji stadij renderiranja, odnosno samo sjenčanje površina koje prikazujemo, se izvodi u zasebnim *device* funkcijama koje poziva glavni kernel. One na neki način predstavljaju naše pixel shadere zaslužne za konačan prikaz svih objekata, što uključuje i teksturiranje. Trenutno se ovdje izvodi jednostavno Phong sjenčanje koje uzima u obzir sva svjetla u sceni bez gubitka svjetline s udaljenosti (praktički, kao da su sva svjelta Sunce).

Za sve implementacijske detalje oko kopiranja geometrijskih podataka u GPU memoriju, alokacije, dealokacije različitih memorija, računanje sjecišta i sl. pogledajte priloženi izvorni kod i popratnu detaljnu PDF dokumentaciju svih metoda koja sadrži i dijagrame klasa.

4.3 Moguće daljnje optimizacije

Ovaj raytracer ima mogućnost renderiranja scene koristeći perspektivnu (prirodnu, poput ljudskog vida) projekciju te ortografsku projekciju. Ta dva načina mijenjaju izračun smjera pojedinih zraka te imaju različita njihova ishodišta, pa je potrebno ovdje imati divergenciju u kodu. Trenutno je to riješeno grananjem u samom kernelu, što nije nužno optimalno budući da dolazi do potrebe za spekulativnim izvođenjem različitih grana dok se ne sazna je li kamera jednog ili drugog tipa, što može dovesti do pretjerane i na kraju nepotrebne potrošnje registara.

Alokacije globalne GPU memorije koje se koriste za iscrtavanje framebuffera prozora mogu biti vrlo promjenjive u određenim situacijama – ako korisnik mijenja rezolucija prozora, podešava različite konfiguracije monitora i sl. To dovodi do potrebe stalne realokacije različite količine memorije, ponekad čak i puno veće od prošlih zahtjeva. Ako se to nastavi događati dovoljno dugo, moguće je da će se GPU memorija značajno fragmentirati i moglo bi čak doći do rušenja programa, stoga trenutno ne preporučamo da se značajno mijenja rezolucija tijekom izvođenja. Bolje rješenje ovog problema bi bilo uvođenje jednog memorijskog bazena koji bi imao pretpostavljeni teoretski maksimum očekivane rezolucije (npr. 6 MP) te dovoljno mjesta za pohraniti toliko piksela. Svaki put kada se promijeni rezolucija, jednostavno bi se iskoristio isti taj bazen, samo s promijenjim vrijednostima rezolucije koje nam govore koji raspon koordinata zaista smijemo koristiti u bazenu. Možebitni nedostatak bi ovdje bilo pretjerano rezerviranje memorije koja neće možda nikada biti u potpunosti iskorištena, zato se može na početku uzeti relativno konzervativna maksimalna rezolucija koja se kasnije u ekstremnim slučajevima može udvostručiti (eksponencijalni rast; po mogućnosti do dane gornje međe), i tako dalje svaki put iznova – na ovaj način niti alociramo prečesto niti rezerviramo previše.

Daljnje potencijalne optimizacije su sve vezane za ubrzavanje najzahtijevnijeg dijela koda – presjecanja zraka s objektima na sceni. Ovaj postupak bi trebao izbjeći što je više posla moguće, i to tipično

primjenom načela „podijeli pa vladaj”. Drugim riječima, želimo da zrake ne moraju za svaki složeni mesh model provjeravati sijeku li se sa svakim od njegovih trokutova ili ne, već da hijerarhijski rade relativno brze provjere. Tako se na prvoj razini hijerarhije treba što brže moći provjeriti prolazi li zraka uopće u blizini modela, jer ako ne prolazi, tada nužno *ne siječe* objekt. Ako ipak siječe tu okolinu, tada se spuštamo na prvu sljedeću nižu razinu hijerarhije i provjeravamo siječe li tamošnju pod-okolinu itd. rekurzivno sve dok ne dođemo do najniže jedinice – trokuta – te se uvjerimo da (ne) siječe konkretan trokut. Ovo se najjednostavnije može implementirati koristeći tzv. *bounding box* tj. kvadre koji omeđuju naše objekte u svjetskom prostoru, no u tom slučaju gubimo fini hijerarhijski prikaz i možemo provjeriti samo najvišu razinu hijerarhije tj. sam kvadar.

Bolji pristup u mnogim situacijama predstavlja *octree* [13], a to je stablo reda 8 gdje čvorovi predstavljaju gore spomenute pod-okoline objekta u hijerarhiji, a korijen predstavlja cijeli objekt. Listovi ovog stabla su pojedinačni trokutovi ili malo veći skupovi trokutova koji se vrlo brzo mogu procesirati za potreban test. Posebnost *octree* strukture je u tome što je vrlo jednostavno provjeriti je li dana točka dio određenog čvora (na bilo kojoj razini) – treba samo znati jednadžbe ravnina kojima je omeđena odgovarajuća kocka tog čvora, a to je uvijek moguće dobiti direktno koristeći činjenicu da znamo korijensku kutiju koja omeđuje korijenski čvor, a *octree* čvorovi predstavljaju pojedine oktante svake od tih kutija rekurzivno, u prirodnom redoslijedu (prvi, pa drugi, treći,..., osmi oktant). Na taj način dobivamo jednadžbe tih 6 ravnina i provjeravamo, koristeći standardni test sa skalarnim produktom, je li taj produkt pozitivan za sve ravnine – ako jest, sigurno smo unutar čvora, a ako nije – nismo. Ovaj test se tada može nastaviti rekurzivno do željene razine detalja – najvjerojatnije do samoga dna.

Osim *octree* pristupa, postoje i drukčiji hijerarhijski pristupi, primjenjivi u različitim situacijama, od kojih su najpoznatiji [13]:

1. grupiranje smisleno povezane geometrije u „sobe” ili ćelije koje tada možemo predstaviti jednostavnom omeđujućom kutijom, s razlikom otprije da je ovoga puta umjetnik/dizajner tipično ručno postavlja i može podešavati tijekom razvoja. Ukoliko struktura prostora ne prati nešto slično ljudskoj arhitekturi (zgradesa sobama koje imaju jasne i male veze između susjednih prostora), ovo nije najbolji pristup jer ne nudi dublju hijerarhijsku podjelu;

2. *k*-d stabla — generalizacija *quadtree*-a, koji predstavljaju slično rješenja *octree*-ima, samo za dvodimenzionalni prostor. *k*-d stabla ne računaju na automatski način gdje će se smjestiti odgovarajući vrhovi kocki u hijerarhiji, već dopušta po mogućnosti optimalan odabir pojedinih vrhova, što dakle dovodi do toga da čvorovi nisu nužno kocke. To može u najboljim slučajevima dovesti do boljih performansi jer se mogu dobiti niža stabla kojima se može brže prošetati. *k*-d stabla su primjenjiva za particioniranje prostora proizvoljne dimenzije, za razliku od *quadtree*-a (samo 2D) i *octree*-a (samo 3D);

3. BSP (engl. *Binary Space Partitioning*) stabla — generalizacija su *k*-d stabla s karakterističnom razlikom da omogućavaju particioniranje ravninama koje nisu nužno poravnate s kanonskim osima. Na ovaj način je moguće jednostavno dijeliti prostor odabirom ravnina koje su komplanarne nekom trokutu od regije koju „režemo”; pritom se mogu pokušati implementirati razne optimizacije poput one da se pokuša odrezati što je manje moguće trokutova (što bi mogao postati problem ukoliko smo primarno ograničeni performansama rasterizatora/mapiranja tekstura).

Još jedan mogući put optimiziranja ovog raytracer-a je uočavanjem činjenice da mi siječemo zrake s objektima u 3D prostoru, a sada smo radili isključivo s 2D mrežom niti na GPU-u. Kada bismo organizirali kernel na način da radi tako da svaka zraka u danom bloku dobije eksplicitan podsegment $[t_0, t_1]$ unutar kojeg smije testirati siječenje, mogli bismo efektivno paralelizirati i provjeru u trećoj dimenziji i dobiti da se cijela dubina (koja je pravilno particionirana u podsegmente), umjesto da se provjerava unutar petlje u svakoj niti, provjerava implicitno zahvaljujući mogućnosti 3D gridova na današnjim GPU-ovima. Ipak, ovakav pristup zahtijeva dosta empirijskog testiranja i finog podešavanja različitih parametara kako bi se dobile optimalne performanse te zahtijeva još dodatnog rada na ovom projektu.

5 Zaključak

U ovom završnom radu smo napravili sažet pregled značajnih algoritama koji se koriste ili su se u značajnijoj mjeri koristili u 3D renderiranju i računalnoj grafici. Fokus je prvo bio na rasterizaciji.

Nastavno na osnovnu matematičku pozadinu i tehnike obrađene u [1], obrazložili smo funkciju modernog grafičkog cjevovoda te pokušali vidjeti kako iskoristiti sposobnosti takvih procesora izlaganjem klasičnog Bresenhamovog algoritma, kao i nekih varijanti s mogućim boljim performansama, zajedno s diskusijom o performansama i implementacijom u pseudokodu. Potom smo izložili napredniji rasterizacijski algoritam koji može u ekstremnim implementacijama koristiti različite arhitekturne detalje radi optimizacije za danu arhitekturu procesora. Naposljetku, predstavili smo relativno najnapredniji i najrecentniji takav algoritam koji cilja omogućiti lakšu paralelizaciju rasterizacije na svim razinama, što bi trebalo omogućiti lakše skaliranje performansi renderiranja na visokoj skali u budućnosti, kako računala postaju sve paralelnija. Tijekom cijelog rada smo pokušali dati suvisle komentare oko performansi dotičnih algoritama te ih kontrastirati s već obrađenim algoritmima kako bi vidjeli koje su moguće prednosti, a koji nedostaci u kontekstu tipa računala koja bi te algoritme trebala u stvarnosti izvoditi.

Na kraju smo se dotakli i raytracinga, uglavnom radi zaokruživanja opće priče o 3D renderiranju. Ovako smo naime, barem na visokoj razini, predstavili sve moguće načine generiranja slika 3D modela na računalu. Raytracer je za kraj zanimljiva tema jer ponavlja neke osnove korištene u rasterizacijskim algoritmima, no za razliku od njih ne zahtjeva pretjerano opširne i specijalno optimizirane algoritme za osnovno generiranje *točnih* i preciznih slika, što je čini popularnom u znanstvenim krugovima čak i kada je višestruko sporija od rasterizacije.

To je čini i idealnom za pokušaj implementacije od nule na grafičkom procesoru koristeći neki od niskorazinskih API-jeva koji nam dopuštaju što je bliži dostup mnogim od procesora na GPU-u — to je upravo napravljeno u priloženom softveru koji prilagođava jedan rudimentaran raytracer za rad na AMD GPU-ovima koristeći ROCm framework. Pritom nije korištena nijedna grafička biblioteka, osim SDL2 za otvaranje prozora na bilo kojoj platformi. Svi detalji oko ovog softvera se mogu pronaći u prilogu, gdje je objašnjeno i kako buildati i koristiti softver.

6 Popis slika

7 Literatura

[1] Mauro Raguzin. *3D renderiranje*, seminar, 2016.

[2] AMD, "RadeonOpenCompute/ROCm – Open Source Platform for HPC and Ultrascale GPU Computing", *AMD*, 2020. [Online].
Dostupno: https://github.com/RadeonOpenCompute/ROCm [Pregledano: 10. 6. 2020.].

[3] Dustin H Land, "I Am Graphics And So Can You :: Part 2 :: Intuition – Faster Than Life", 2017. [Online].
Dostupno: https://www.fasterthan.life/blog/2017/7/11/i-am-graphics-and-so-can-you-part-2-intuition [Pregledano: 10. 10. 2019.].

[4] Khronos Group, "OpenGL Shading Language - OpenGL Wiki", *Khronos Group* 2017. [Online].
Dostupno: https://www.khronos.org/opengl/wiki/OpenGL_Shading_Language [Pregledano: 10. 10. 2019.].

[5] Mike Bailey, "Push Constants – Vulkan", *Oregon State University*, 2018. [Online].
Dostupno: http://web.engr.oregonstate.edu/~mjb/vulkan/Handouts/PushConstants.1pp.pdf [Pregledano: 10. 10. 2019.].

[6] Ivan Sutherland i Gary W. Hodgman, "Reentrant Polygon Clipping", *Communications of the ACM*, vol. 17, 1974.

[7] Chris Hecker, "Perspective Texture Mapping", *Game Developer Magazine*, UBM Tech, 1995-1997.

[8] Morgan McGuire, "A Game Developer's Perspective of SIGGRAPH 2001", *flipCode - Game Development News & Resources*, 2001. [Online] Dostupno: http://www.flipcode.com/misc/siggraph2001.shtml [Pregledano: 10. 10. 2019.].

[9] Mats Byggmastar, "Fast affine texture mapping (fatmap.txt)", *Flipcode*, 8. 7. 1996. [Online].
Dostupno: http://www.flipcode.com/documents/fatmap.txt [Pregledano: 10. 10. 2019.].

[10] Michael D. McCool, Chris Wales i Kevin Moule, "Incremental and Hierarchical Hilbert Order Edge Equation Polygon Rasterization", *Computer Graphics Lab, Department of Computer Science, University of Waterloo*, 2001. [Online] Dostupno: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.18.5738&rep=rep1&type=pdf [Pregledano: 11. 6. 2020.].

[11] David Kanter, "Radeon – Dissecting the Polaris Architecture", *AMD* [Online].
Dostupno: https://www.techpowerup.com/gpu-specs/docs/amd-polaris-architecture.pdf [Pregledano: 10. 10. 2019.].

[12] P. Shirley, S. Marschner, M. Ashikhmin, M. Gleicher, N. Hoffman, G.t Johnson, T. Munzner, E. Reinhard, K. Sung, W. B. Thompson, P. Willemsen i B. Wyvill, "Fundamentals of Computer Graphics". Natick, MA, USA: A. K. Peters, Ltd. 2009.

[13] S. R. Buss, "3D Computer Graphics: A Mathematical Introduction with OpenGL". New York, NY, USA: Cambridge University Press, 2003.

8 Prilozi

Sastavni dio ovog završnog rada je raytracer specifično napisan za AMD GPU-ove koristeći ROCm framework. Upute o kompajliranju i preporučenoj konfiguraciji možete pronaći u datoteci `README` u glavnom direktoriju dobivenom raspakiravanjem priložene ZIP datoteke.

Postoje dvije ZIP datoteke — koju ćete odabrati ovisi o hipcc verziji koju imate instaliranu i koja vam u potpunosti ispravno funkcionira. `raytracer_hip2` je jedina u potpunosti službeno podržana verzija budući je jedino mogla biti u potpunosti testirana od strane autora. Tu verziju koristite ako imate hipcc stariji od glavne verzije 3.

Inače, koristite alternativni kod u `raytracer_hip3` koji je organiziran tako da zaobilazi neke uočene probleme s linkerom koje je ova nova verzija prouzročila na autorovom stroju.