UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Tomaž Kunaver

Izločanje nevidnih ploskev v 3D-pogonih

DIPLOMSKO DELO
NA UNIVERZITETNEM ŠTUDIJU

Mentor: prof. dr. Saša Divjak

Ljubljana, 2010

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Namesto te strani **vstavite** original izdane teme diplomskega dela s podpisom mentorja in dekana ter žigom fakultete, ki ga diplomant dvigne v študentskem referatu, preden odda izdelek v vezavo!

IZJAVA O AVTORSTVU

diplomskega dela

Spodaj podpisani/-a Tomaž Kunaver,

z vpisno številko 63030306,

sem avtor/-ica diplomskega dela z naslovom:

Izločanje nevidnih ploskev v 3D-pogonih

S svojim podpisom zagotavljam, da:

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

V Ljubljani, dne 08.09.2010 Podpis avtorja/-ice:

Kazalo

Seznam uporabljenih kratic in simbolov

- PVS - “Potentially visible set”. Nabor (verjetno) vidnih primitivov.
- AABB - “Axis aligned bounding box”. Osno poravnan vseobsegajoči kvader.
- FPS - “Frames per second”. Število sličic na sekundo.
- Octree - Osmiško drevo.
- Ocnode - Vozlišče v osmiškem drevesu.
- Frame - Sličica. Izraz uporabljamo, ko govorimo o zaslonski sliki, ki jo izrisujemo.
- Frame rate - Število izrisanih sličic na sekundo.
- Pixel - zaslonska točka.
- Occlusion query - poizvedba zakritosti.
- Z-buffer - metoda, običajno implementirana na grafični kartici, ki preprečuje, da bi oddaljeni primitivi prerisali bližnje. S tem imenom označujemo tudi medpomnilnik na grafični kartici, ki hrani globinske vrednosti zaslonskih pik.
- GPE - Grafična procesna enota.
- CPE - Centralna procesna enota.
- Occluder - zakrivalna ploskev (v splošnem objekt).

Povzetek

V diplomski nalogi sem raziskal problem izločanja nevidnih ploskev v 3D-pogonih ter implementiral štiri metode izločanja. Pogoni v današnjih igrah za izrisovanje uporabljajo bolj in bolj zmogljive grafične kartice, ki lahko sedaj pomagajo tudi pri izločanju zakritih ploskev, na primer z uporabo poizvedb zakritosti ("occlusion queries"). Pogosto pa je ozko grlo v komunikaciji med CPE in GPE veliko število primitivov, ki jih pošiljamo GPE. Zato je pomembno izločiti čim več geometrije že v samem pogonu, na strani CPE. Na izrisani sliki je v vsakem trenutku lahko vidnih na milijone poligonov, običajno pa je še več takih, ki so z drugimi poligoni zakriti ali niso vidni, ker so zunaj pogleda kamere ali pa so obrnjeni stran od opazovalca. Metode, ki sem jih implementiral, izločajo tako nevidne poligone (poligone zunaj piramide pogleda ter poligone, ki so obrnjeni stran od opazovalca) kot zakrite poligone. V ta namen sem implementiral metodo sevanja žarkov ter dve preprosti različici metode, ki za izločanje uporablja poizvedbe zakritosti. Pri tem sem izvedel tudi različne meritve in analiziral učinkovitost izvedb posameznih metod. Izkaže se, da največje pohitritve dosežemo z vnaprejšnjim računanjem vidljivosti, kar pa se v praksi izkaže za računsko zelo zahteven problem.

Ključne besede:

izločanje nevidnih ploskev, 3D-pogon, poizvedbe zakritosti, sevanje žarkov, piramida pogleda

Abstract

In the following diploma thesis I have researched the problem of hidden surface determination in 3D engines and implemented four methods of culling. Engines in modern 3D games use more and more powerful graphics cards for rendering, which can now help with occlusion culling as well, for example by providing mechanism called occlusion queries. The bottleneck in communication between CPU and GPU most often lies in sending large numbers of graphics primitives. That is why it is vital to cull away as much geometry as possible on the CPU side. In a rendered image millions of polygons may be visible each frame, but usually even more are occluded by other polygons or are invisible because they lie outside of the view frustum or are facing towards the viewer with their back sides. The methods which I have implemented cull away invisible polygons (those outside the view frustum and those facing away from the viewer) as well as occluded ones. For this purpose I implemented raycasting method and two simplified versions of a method, that use occlusion queries for culling. I have also performed various experiments and analyzed efficiency of different implementations. It turns out that we can achieve highest speedups by precomputing visibility, which however turns out to be computationally a very difficult problem.

Key words:

occlusion culling, 3D engine, occlusion queries, raycasting, view frustum

Poglavje 1

Uvod

V pričujočem diplomskem delu sem se ukvarjal z metodami za izločanje nevidnih ploskev v 3D-pogonih. V ta namen sem raziskal in implementiral štiri različne metode izločanja. Dandanes večino grafičnega preračunavanja opravijo grafične kartice, ki zmorejo v sekundi izrisati že nekaj sto milijonov trikotnikov. V 3D pogonih pa ozko grlo pogosto predstavlja komunikacija med procesorjem (CPE) in grafično kartico (GPE). Za vsako sličico, ki jo izrišemo, moramo GPE poslati seznam primitivov (ali indekse na primitive), ki naj jih izriše. Z večanjem števila trikotnikov se veča tudi količina podatkov, ki jih pošiljamo GPE. Čeprav zmore GPE procesirati veliko količino primitivov na sekundo in to običajno ne predstavlja ozkega grla, pa je pomembno, da na GPE ne pošljemo odvečne geometrije, t.j. geometrije, ki v končni fazi ne bo prispevala k izrisani sliki, saj s tem upočasnjujemo komunikacijo med CPE in GPE, kar pa pogosto predstavlja ozko grlo. Zato je pomembno, da že na strani CPE (v samem 3D-pogonu) izločimo čim več nevidne geometrije. Taka geometrija so recimo ploskve ali objekti, ki se nahajajo izven piramide pogleda kamere ter ploskve, ki so zakrite od drugih ploskev.

V namen testiranja postopkov izločanja sem uporabil lasten 3D-pogon "Digizet", ki sem ga na kratko opisal v posebnem razdelku. Digizet sem razvil v teku študija, primarno kot seminarsko nalogo za predmet Računalniška grafika. Izkazalo se je, da je implementacija 3D-pogona prevelik zalogaj, da bi ga implementiral v okviru seminarske naloge, zato sem si del implementacije pustil za kasneje. Najbolj pomembna stvar, ki je pogonu še manjkala, je bilo izločanje nevidnih ploskev. Zato sem si kot cilj te diplomske naloge zadal raziskati to področje ter implementirati in primerjati nekaj konkretnih metod izločanja.

Izločanje nevidnih ploskev je široko področje in običajno v 3D-pogonih predstavlja kompleksen problem. Danes si moderne, realnočasovne 3D-igre

brez izločanja nevidnih ploskev ne moremo več zamisliti. Kljub temu, da gre veliko časa in truda v implementacijo postopkov za izločanje, pa samo izločanje na vizualno podobo igre ne vpliva. Če izločanja ne bi implementirali, bi igra izgledala popolnoma enako in uporabnik ne bi opazil ničesar. Edina razlika, ki jo izločanje prinese, je pohitritev pogona. Z izločanjem lahko pogon bistveno pohitrimo, kot bomo videli v naslednjih poglavjih. Zato implementacije tehnik izločanja v modernem 3D-pogonu ne smemo zanemariti.

1.1 Predstavitev Digizeta

Digizet je 3D-pogon napisan v jezikih Java in LUA. Jezik LUA sem uporabil za krmiljenje pogona: nalaganje modelov, branje/shranjevanje nastavitev, kontrolo uporabniškega vmesnika ipd. Pretežni del pogona pa je napisan v Javi. Digizet sem začel razvijati skupaj z dvema sošolcema v okviru seminarske naloge pri predmetu Računalniška grafika. Od takrat je doživel že mnogo posodobitev in izboljšav. Med razvijanjem ogrodja za seminarsko nalogo se je izkazalo, da je problem izločanja tako zahteven, da bi zanj potreboval še enkrat toliko časa, kot za preostali del pogona. Zato sem se tega kompleksnega problema lotil v okviru diplomske naloge.

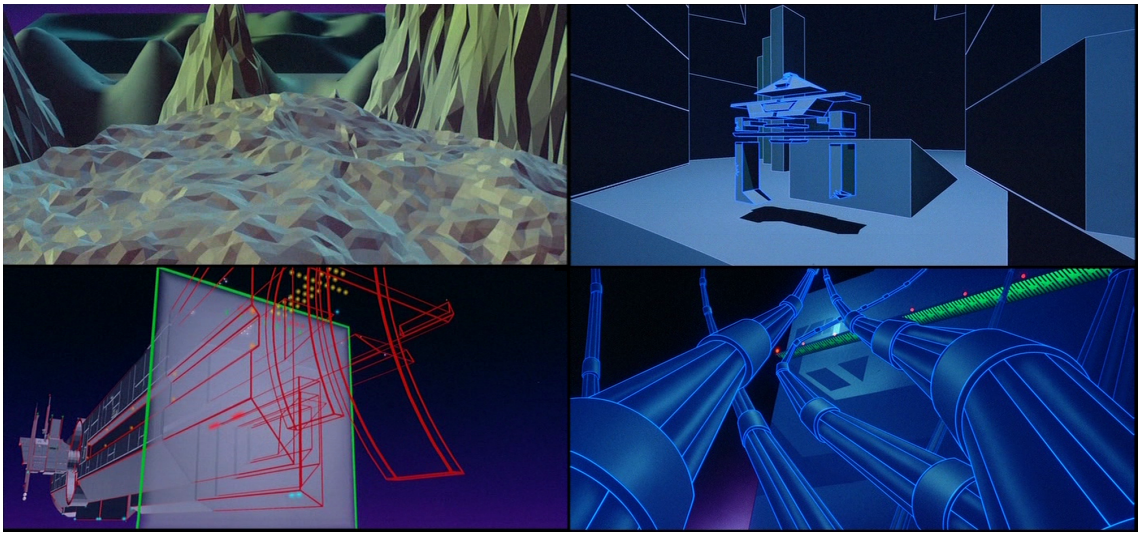
Digizet je bil zamišljen kot preprost pogon, ki bi prikazoval in omogočal interakcijo s prostranimi, digitalnimi okolji, podobnimi tistim iz filma Tron. Primer takega okolja prikazuje slika 1.1. Zaradi specifičnosti okolja je bilo treba tudi pogon načrtovati tako, da bi izkoristil lastnosti okolja. Ena takih lastnosti je, da okolja v Tronu vsebujejo malo ali nič tekstur. Druga lastnost je, da so okolja preprosta in oglata, brez krivulj ter izredno prostrana.

Za opisan tip scene se je izkazalo, da je osmiško drevo dobra izbira za graf scene, predvsem zato, ker je primeren za prostrana okolja, saj ima faktor vejanja večji kot nekatera druga drevesa (glej poglavje 2). Poleg tega so okolja statična in odprta, kar vpliva na izbiro metode za izločanje. Mislimo si, da realnočasovne metode ne bodo dobro izkoristile lastnosti takega okolja. Glede na to, da scene v Digizetu vsebuje malo dinamične geometrije, bi bila izbira predprocesiranja vidljivosti bolj primerna.

Digizet je sestavljen iz večih delov:

- Sistem za detekcijo in odziv na trke ("Collision detection and response"). V Digizetu preverjamo trke med svetom in likom, ki ga igralec upravlja, diski, delci, itd.

- Sistem delcev ("Particle system"). Iskre, ki se krešejo ob trkih med



Slika 1.1: Primeri scen iz filma Tron. Geometrija je preprosta, oglata, vsebuje malo tekstur in scene so zelo prostrane. Vse to vpliva na načrtovanje pogona in tudi na izbiro ustrezne metode za izločanje.

diskom in svetom. Tudi tu moramo uporabiti detekcijo s svetom in pravilno simulirati odboj od tal ipd.

- Periferni sistemi. Sem lahko štejemo nalagalnik objektov, grafični vmesnik (meniji, konzola, ...), integracijo skriptnega jezika itd.

- Izrisovalnik. Najbolj pomemben del pogona. Pri izrisovanju uporabljam več različnih metod za pohitritev, na primer uporabo ogliščnih medpomnilnikov ("Vertex Buffer Objects"), indeksna polja, prikazovalne sezname ("Display lists"), teksturna vedra ("Texture buckets") itd. Predvsem uporaba teksturnih veder je zelo vplivala na implementacijo izločanja, saj je bilo potrebno trikotnike iz drevesa izbrati na tak način, da so ohranila pravilen vrstni red znotraj veder. V nasprotnem primeru bi lahko z izločanjem pogon upočasnili, saj bi izničili efekt optimizacije s teksturnimi vedri.

Pomembna lastnost pogona, ki jo moramo upoštevati tudi pri implementaciji izločanja, je ta, da vsebuje zelo malo dinamične geometrije. To upoštevamo pri izbiri drevesa za graf scene ter izločanju.

Poglavje 2

Graf scene (Scene graph)

Pod sceno razumemo vso geometrijo, ki jo najdemo v našem trodimenzionalnem svetu. Ta geometrija je lahko statična (se med izvajanjem programa ne spreminja) ali dinamična (spreminja obliko ali lego, orientacijo itd.). Kadar je scena zelo velika, je tudi njeno upodabljanje počasno. Ker pa običajno ne izrisujemo celotne scene naenkrat, ampak le njen del, je pomembno, da so objekti v sceni hierarhično urejeni. Tako lahko na hiter način naredimo izbor geometrije iz scene, ki jo nameravamo izrisati.

Pri hierarhični ureditvi mislimo na prostorsko ureditev. Najbolj pogosto v ta namen uporabljamo različne drevesne strukture. V tem poglavju si bomo pogledali tri najbolj pogosto uporabljane strukture: BSP-drevo, osmiško drevo in kd-drevo. Osmiško drevo, ki sem ga implementiral tudi v Digizetu, si bomo pogledali bolj podrobneje in tudi pogledali različne probleme in trike pri gradnji ter uporabi tega drevesa. Izbira drevesa ni preprost problem – če hočemo za dani problem izbrati najbolj primerno drevo, moramo najprej dobro poznati posamezna drevesa ter njihove slabosti in prednosti v različnih situacijah.

Drevo je sestavljeno iz vozlišč, pri čemer so nekatera končna vozlišča ("leaf nodes"), ki ponavadi vsebujejo geometrijo (točke, trikotnike, informacijo o teksturah, ...), druga pa vmesna vozlišča, ki praviloma ne vsebujejo geometrije, ampak so namenjena za podporne elemente drevesa. Obstaja posebno vozlišče, t.i. korensko vozlišče ("root node"). Vsa vozlišča, razen korenskega, imajo natanko enega starša, ki je tudi vozlišče, ter poljubno število otrok. Vozlišča brez otrok so listi drevesa (končna vozlišča). Namesto dreves lahko uporabimo tudi usmerjene aciklične grafe ([3, 9]).

2.1 BSP-drevo (Binary space partitioning tree)

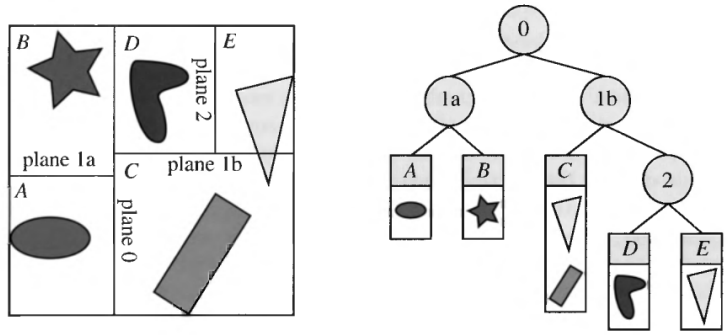
BSP-drevo je najbolj uporabljana struktura za graf scene v 3D-igrah. Prvič je bila uporabljena v prvoosebnih streljankah v začetku devetdesetih, od takrat naprej pa je uporabljena v večini iger ([4]). Dandanes se uporabljajo tudi nekatere modernejše hibridne strukture, saj imajo BSP-drevesa določene slabosti (v izvajalnem času jih je težko spreminjati, delitev prostora gre počasi – binarno, ...), še zmeraj pa je odločitev za BSP-drevo v današnjih igrah pogosta.

Binarno iskalno drevo bi lahko definirali kot običajno binarno drevo, ki ga uporabljamo za iskanje politopov v n-dimenzionalnem prostoru, v našem primeru za iskanje poligonov v trodimenzionalnem prostoru. Drevo predstavlja celoten prostor, posamezno vozlišče v drevesu pa predstavlja določen konveksni podprostor ([2]). Drevo predstavlja hierarhično delitev n-dimenzionalnega prostora v konveksne podprostore, pri čemer v vsakem vozlišču hranimo hiperravnino, ki prostor razdeli na dve polovici (dva polprostora) ter množico politopov, ki pripadajo posameznemu podprostoru. Delitev prostora na dva polprostora rekurzivno ponavljamo.

V računalniški grafiki obstajata dva tipa BSP-dreves, in sicer osno poravnana ("axis-aligned") in poligonsko poravnana ("polygon-aligned"). Poglejmo si primer prvega v dvodimenzionalnem prostoru, kjer je hiperravnina premica. Na sliki 2.1 smo prostor najprej razdelili navpično po premici 0 ("plane 0"), kar nam je dalo dva polprostora – prvega, ki obsega podprostora B in A, in drugega, ki obsega podprostore D, E in C. V naslednjem koraku smo prvi polprostor naprej razdelili z vodoravno premico ("plane 1a") in dobili podprostora A in B, drugega pa z drugo vodoravno premico ("plane 1b"), ki nam je dal podprostora C in unijo med D in E. V zadnjem koraku smo razdelili še podprostor iz prejšnjega koraka (unija med D in E) na dva nova podprostora D in E. S tem smo postopek končali. V vozliščih drevesa hranimo informacijo o hiperravninah, ki smo jih uporabili za deljenje prostora, v vozliščih drevesa pa like, ki delno ali v celoti spadajo v podprostor, ki ga vozlišče predstavlja.

V trodimenzionalnem primeru bi za hiperravnine vzeli navadne ravnine. Postopek gradnje osno poravnanega BSP-drevesa vedno začnemo z AABB ("axis-aligned bounding box"), ki obsega vso geometrijo, ki jo želimo predstaviti z drevesom. Nato rekurzivno delimo prostor na dva manjša podprostora. Tu se pojavi več vprašanj:

1. Kako izbrati ravnino, ki prostor razdeli na dva polprostora?
2. Kdaj ustaviti rekurzijo?



Slika 2.1: Primer osno poravnanega BSP-drevesa. S črkami A do E so označeni končni (najmanjši) podprostori, ki sovpadajo z listi drevesa. Ker je trikotnik na sliki na meji med podprostoroma E in C, ga štejemo k obema. Slika povzeta po [9], stran 651.

3. Kaj narediti z objekti, ki se nahajajo na meji dveh ali večih podprostorov?

Odgovor na prvo vprašanje vključuje več različnih rešitev. Pri kd-drevesih, ki so poseben primer BSP-dreves, za delitev izmenično uporabljamo osi x, y in z. Druga možnost je, da prostor zmeraj delimo vzdolž ravnine, ki seka največjo ploskev AABB-ja tega podprostora. Če je na primer najdaljša stranica AABB-ja v smeri osi x, potem postavimo delilno ravnino v neki točki x0, pri čemer je vzporedna z YZ ravnino. Poleg delilne ravnine pa moramo izbrati tudi točko, v katero postavimo ravnino. To je kompleksen problem in na tem mestu bomo omenili le nekaj možnosti. Ena je ta, da poskušamo deliti tako, da pri deljenju razrežemo kar najmanj objektov. Pri tem iščemo delilno točko tako, da oba polprostora vsebujeta približno enako število objektov. Drugi možen kriterij pa je tudi čim manjša razlika v velikostih polprostorov.

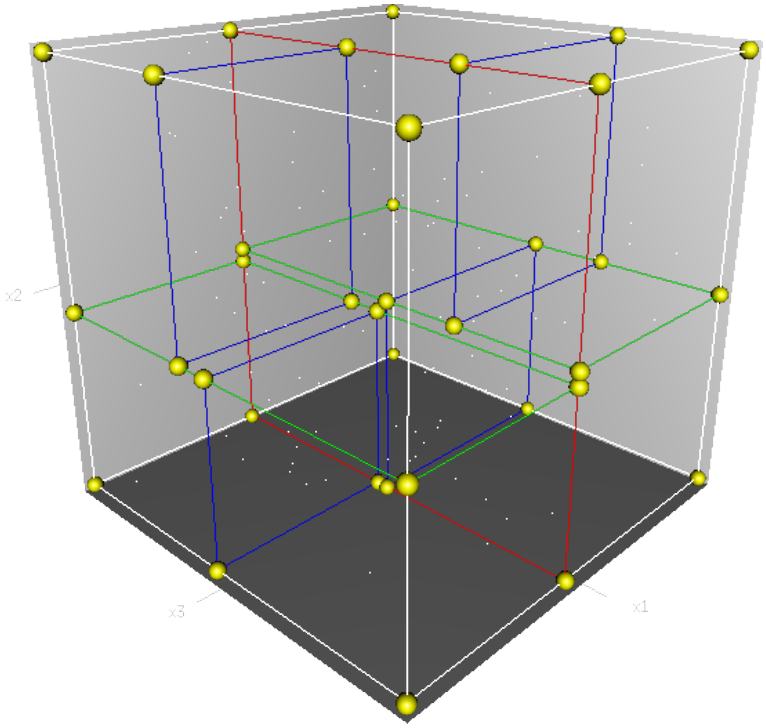
Odgovor na drugo vprašanje je odvisen od tega, za kaj bomo uporabljali BSP-drevo. Če želimo natančno sortiranje objektov po oddaljenosti, moramo rekurzijo ponavljati toliko časa, dokler ne dobimo podprostorov s samo enim objektom. Če je v listu drevesa več objektov, ti očitno niso sortirani po oddaljenosti (sortiranje pravzaprav dosežemo z rekurzivnim obiskovanjem vozlišč drevesa. Objekte v listu drevesa bi lahko sortirali tudi ročno in tako zagotovili željen vrstni red objektov). Vendar pa običajno zadostuje že delna urejenost. Delitev lahko ustavimo takrat, ko število objektov v listu doseže določeno mejo. Pri tem lahko rekurzijo omejimo tudi z največjo dovoljeno globino drevesa in drugimi kriteriji.

Tretje vprašanje nam ponuja več rešitev. BSP strukturi lahko na primer dovolimo, da hrani objekte tudi v nekončnih vozliščih. Tako objekt shranimo tako visoko v hierarhiji, da pripada samo enemu podprostoru. To je zmeraj mogoče – v najslabšem primeru moramo objekt shraniti v korensko vozlišče. Prednost tega pristopa je v tem, da v drevesu hranimo le eno kopijo objekta. Prinaša pa ta pristop druge slabosti, ki jih tu ne bomo obravnavali. Druga možnost pa je, da objekt shranimo v vozliščih vseh tistih podprostorov, ki jih seka. Slaba stran tega je podvajanje objekta, vendar pa to ne sme biti prevelik problem, saj v vozliščih hranimo le reference na objekt. Kot bomo videli pri osmiških drevesih, je ta pristop najbolj smiseln. Tretja možnost pa je, da objekt razrežemo po hiperravninah, ki delijo podprostore in posamezne dele objekta dodelimo ustreznim podprostorom. Ta pristop se uporablja pogosto, vendar pa ima to slabost, da iz enega objekta dobimo več samostojnih objektov, ki jih moramo v celoti hraniti v pomnilniku (in ne le reference na isti objekt). Pri osmiških drevesih bomo videli, da je to še večji problem, saj tam delitvenih ravnin ne moremo poljubno izbirati in zato pogosto prihaja do rezanja objektov in s tem ustvarjanja velikega števila novih objektov.

Pri poligonsko poravnanih BSP-drevesih velja večino stvari, ki smo jih že povedali. Razlika pri tem tipu dreves je v tem, da tokrat delilna ravnina ni poravnana z osmi sveta, temveč s poligonom, ki ga izberemo kot delilni poligon. Vsi objekti, ki so na eni strani ravnine poligona, spadajo v en polprostor, ostali pa v drugega. Pri tem objekte, ki sekajo ravnino poligona, razrežemo na dva dela. Pri teh drevesih dobimo točen “front to back” ali “back to front” vrstni red, kar nam pride prav, ko želimo objekte izrisati v takem vrstnem redu, da najbolj oddaljenim sledijo manj oddaljeni (Slikarjev algoritem). Pri tem ne potrebujemo tehnike “Z-buffer”, saj se poligoni ne prekrivajo. Tu se nam postavlja pomembno vprašanje: kateri poligon izbrati za delitev? Strategij za izbiro delilnega poligona je veliko. Na splošno je dobro, da izbiramo take poligone, ki ustvarijo čim bolj uravnoteženo drevo. Vsak poligon bi moral prostor razdeliti na dva polprostora, ki vsebujeta približno enako število objektov. Ekstremni primer bi bil, ko poligon razdeli prostor na en prazen polprostor in na drug polprostor, ki vsebuje vse ostale objekte. S tem ne pridobimo ničesar ampak še izgubimo na hitrosti, saj moramo iterirati skozi vozlišča, ki nam ne prinesejo nič “novega”.

2.2 kd-drevo (k-dimensional tree)

kd-drevesa so še ena od podatkovnih struktur za prostorsko particioniranje in so pravzaprav poseben primer BSP-dreves. V splošnem se uporabljajo za iskanje po podatkih z večdimenzionalnimi ključi (iskanje po principu najbližjega soseda, [13]) ipd. Podobno kot BSP-drevo je tudi kd-drevo binarno drevo, kjer vsako nekončno vozlišče razdeli prostor na dva polprostora. Vsako tako vozlišče definira hiperravnino, poravnano z osmi sveta, in razdeli objekte na tiste, ki so na eni strani hiperravnine, in na druge, ki so na drugi strani. Primer drevesa vidimo na sliki 2.2.



Slika 2.2: Primer kd-drevesa. Prva delitev, ki razdeli korensko vozlišče, ki je pobarvano z belo barvo, je označena z rdečo. Dobimo dva polprostora, ki jih naprej razdelimo po zelenih oznakah. Zadnje štiri podprostore razdelimo po modrih oznakah in tako dobimo končnih 8 podprostorov, ki predstavljajo liste kd drevesa. Slika povzeta po [13].

Delilne ravnine lahko izberemo na veliko različnih načinov, tako da dobimo veliko različnih dreves. Običajno pa se držimo naslednjih dveh omejitev pri gradnji kd-drevesa:

1. Pri prvi delitvi izberemo delitveno ravnino vzdolž x osi, kar pomeni, da je vzporedna z YZ ravnino. Pri naslednjem nivoju izberemo delilno ravnino vzdolž y osi, na naslednjem pa vzdolž z osi. Na četrtem nivoju začnemo zopet od začetka – najprej izberemo ravnino vzdolž x osi, nato vzdolž y, in tako naprej.

2. Pri vsaki delitvi izberemo točko na osi, vzdolž katere želimo postaviti delitveno ravnino, in sicer tako, da izberemo mediano točk glede na koordinato, vzdolž osi katere delimo. Če na primer postavljamo delitveno ravnino vzdolž osi x, bomo izračunali mediano vseh x koordinat od točk, ki so v tem prostoru. To nam zagotavlja, da dobimo uravnoteženo drevo, kar je običajno želena, ne pa nujno potrebna lastnost. V nekaterih primerih namreč taka izbira ni optimalna ([13]).

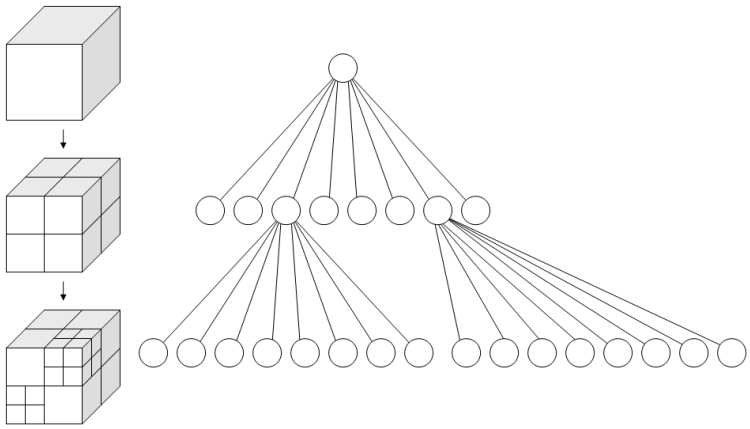
Na kd-drevesa lahko gledamo kot na vmesno različico med BSP in osmiškimi drevesi. Na eni strani so sorazmerno preprosta za gradnjo in uporabo (vse delilne ravnine so poravnane z osmi sveta), po drugi strani pa lahko lego delilnih ravnin poljubno določimo, kar osmiško drevo ne dopušča. Tako se lahko bolje prilagodimo danim podatkom. Še zmeraj pa je spreminjanje kd-dreves v izvajalnem času relativno zapleteno in s tem počasno, kar na primer pri osmiških drevesih ni problem. Draga je tudi gradnja teh dreves. Po drugi strani pa se danes kd-drevesa zelo pogosto uporabljajo pri metodi sledenja žarkom ("ray tracing"). Avtor članka [12] trdi, da so kd-drevesa najboljša izbira za gradnjo hitrih sistemov, ki delujejo na principu sledenja žarkom. Tam je pomembno predvsem, da znamo drevo hitro obhoditi.

2.3 Osmiško drevo (Octree)

Osmiška drevesa so konceptualno najlažja oblika strukture za urejanje geometrije v hierarhično ureditev od obravnavanih treh. Zato sem se tudi odločil za njihovo implementacijo v Digizetu. Tej odločitvi je botrovalo tudi dejstvo, da so prav osmiška drevesa najbolj primerna za velike scene na prostem ("outdoor environments"), kar sem v Digizetu tudi skušal doseči. Za razliko od binarnih dreves namreč tu na vsakem koraku prostor razdelimo na osem podprostorov namesto dveh, zato rekurzija veliko hitreje pride do dna. Pri velikih scenah je to zelo pomembno, saj lahko premikanje po drevesu hitro postane ozko grlo. Ker so osmiška drevesa tako preprosta, je mogoče veliko operacij nad njimi zelo pohitriti. Njihova slabost pa je v tem, da pri izbiri delitvenih ravnin pravzaprav nimamo dosti možnosti. Izbiramo lahko le, ali bomo vozlišče delili

še naprej ali ne. S tem se drevo geometriji ne more dobro prilagoditi, kakor se lahko na primer BSP-drevo, pri katerem si prizadevamo za to, da bi bilo čimbolj uravnoteženo.

Primer osmiškega drevesa prikazuje slika 2.3. Slika 2.4 kaže primer razdelitve prostora z osmiškim drevesom na konkretnem modelu.



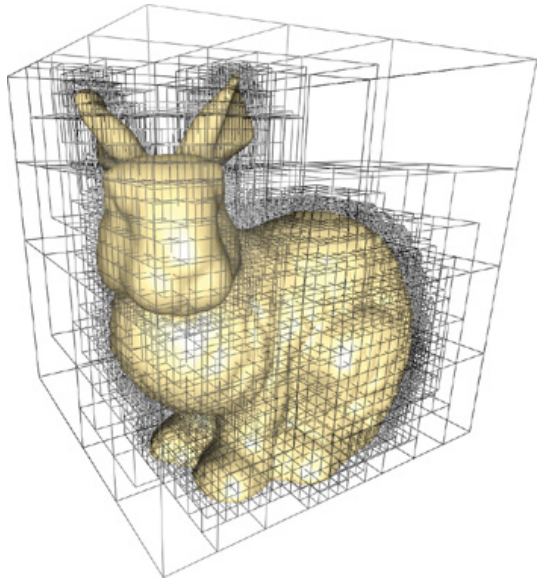
Slika 2.3: Skica rekurzivnega deljenja osmiškega drevesa. Začnemo v korenskem vozlišču, ki ga razdelimo v osem oktantov. Oktant nato rekurzivno naprej razdelimo v osem oktantov. Desno je prikaz osmiškega drevesa kot podatkovne strukture.

Ekvivalent osmiškemu drevesu v 2D je štiriško drevo ("quadtree"). Primer štiriškega drevesa kaže slika 2.5. V primeru na sliki smo si za cilj vzeli takšno razdelitev, da noben list drevesa ne vsebuje več kot en objekt, dovolimo pa, da so listi prazni. Po vrsti sledijo modra, rdeča, zelena in rjava delitev. V vsakem obhodu rekurzijo najprej pogledamo, koliko objektov je znotraj kvadrata. Če je to število večje od 1, kvadrat rekurzivno razdelimo na 4 enake kvadrate in postopek ponovimo na otrocih. Rekurzijo ponavljamo toliko časa, dokler ni pogoj o največ enem objektu v kvadratu izpolnjen.

2.3.1 Gradnja osmiškega drevesa

V tem razdelku si bomo bolj natančno pogledali, kako poteka gradnja drevesa in kako konkretno je implementiran v Digizetu.

Gradnjo začnemo v korenskem vozlišču, ki mora obsegati vso geometrijo v svetu. Temu v splošnem pravimo tudi AABB ("Axis aligned bounding box"). Pri osmiškem drevesu ima AABB obliko kocke in je seveda vedno poravnan



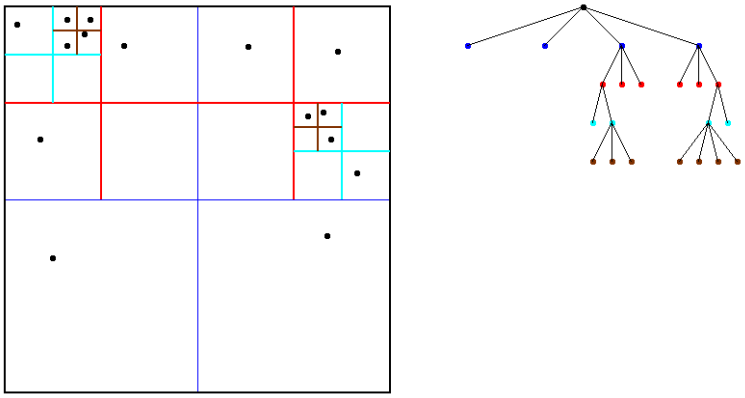
Slika 2.4: Primer razdelitve prostora z osmiškim drevesom na primeru konkretnega modela. Slika last podjetja Nvidia: http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter37.html

z osmi sveta. Velikost korenskega AABB-ja je lahko vnaprej podana (v Digizetu recimo 4096 dolžinskih enot. V Digizetu je ena enota 1 meter v "realnem svetu") ali pa se izračuna ob gradnji drevesa – to naredimo tako, da pogledamo skrajne koordinate x, y in z od dane geometrije. Radij AABB-ja tako postane največja od teh treh koordinat po absolutni vrednosti. Gradnjo drevesa začnemo tako, da nad korenskim vozliščem poženemo rekurzijo, ki za argument sprejme objekt, ki bi ga radi dodali v drevo. Rekurzija v enem ciklu doda objekt osmiškemu vozlišču. Če je s to akcijo število objektov v vozlišču zraslo čez določeno mejo, poskusi vozlišče naprej razdeliti na osem otrok in požene rekurzijo za vsakega posebej, tako da ga vstavi v ustreznega otroka. Vozlišča ne sme razdeliti v primeru, ko je globina vozlišča večja od maksimalne dovoljene. Tu se srečamo z več problemi oz. vprašanji:

1. Kako določiti, kateremu oktantu pripada objekt?

2. Kakšno maksimalno globino naj izberemo?

3. Kakšno maksimalno omejitev naj izberemo za število objektov v listu drevesa?



Slika 2.5: Štiriško drevo, kjer v vsakem listu drevesa hranimo največ en objekt. Delitev smo začeli v največjem kvadratu (črna barva) in ga razdelili na 4 kvadrante (modra barva). Potem smo zgornja kvadranta naprej razdelili na 2 manjša (rdeča barva). Postopek smo rekurzivno ponavljali, dokler nismo v vsakem kvadratu dobili kvečjemu 1 objekt.

Za vsako vprašanje si bomo pogledali mogoče rešitve oz. najboljšo možno izbiro.

1. To je zelo pomembno vprašanje, saj lahko odgovorimo na več različnih načinov, od katerih je odvisna nadaljna raba drevesa. Problem se namreč pojavi pri objektih, ki so na meji med dvema ali več (največ osem) oktanti. Ta problem sem v Digizetu poskusil rešiti na tri načine, od katerih v praksi uporabljam le dva:

   (a) Problemu mejnih objektov se lahko izognemo tako, da za dani objekt izračunamo središče (v Digizetu, kjer imamo opravka le s trikotniki, izračunamo težišče trikotnika) in pogledamo, v katerem oktantu je ta točka. Objekt dodelimo temu oktantu in prezremo dejstvo, da je deloma prisoten tudi v drugih. Ta rešitev je na splošno najslabša, saj zelo omeji uporabo drevesa za izrisovanje geometrije pa tudi za izločanje nevidnih ploskev. Pri običajni rabi drevesa za izrisovanje lahko pride do grafičnih artefaktov, kot so izginjajoči trikotniki in podobno.

   (b) Druga možnost je, da trikotnik razrežemo na več manjših trikotnikov tako, da ti trikotniki v celoti pripadajo le enemu oktantu. To rešitev si bomo pogledali v posebnem razdelku.

(c) Tretja možnost pa je ta, da referenco na objekt shranimo v vse oktante, ki jih seka objekt. Ta rešitev se izkaže za daleč najboljšo. Vendar pa pri izrisovanju pride do problema podvojenih trikotnikov. Ker je trikotnik prisoten v več oktantih, ga izrišemo večkrat. Vendar pa se da temu problemu izogniti. V Digizetu hranim tabelo z IDji vseh trikotnikov v svetu skupaj z resničnostno zastavico, ki pove, ali je bil v trenutni sličici trikotnik že izrisan. Če je bil, ga ne izrišemo znova. Ker pa je trikotnikov veliko je tudi tabela zelo velika, kar pomeni veliko zgrešitev v pomnilniku pri dostopu do tabele. Do nje pa moramo dostopati pri vsakem trikotniku, ki ga želimo izrisati. Kljub temu problemu se izkaže, da je ta rešitev najboljša kar se tiče hitrosti izrisovanja.

(d) Omeniti moramo še eno možnost, ki je v Digizetu nisem implementiral, vendar se tudi uporablja. Dovolimo namreč lahko, da se objekti shranjujejo tudi v nekončnih vozliščih. Podobno rešitev smo omenili že pri BSP-drevesih. Če objekt namreč shranimo tako visoko v hierarhiji, da ne seka nobenega drugega podprostora (to je zmeraj mogoče – v najslabšem primeru ga shranimo v korenskem vozlišču, ki zagotovo obsega vse predmete v svetu), smo ta problem s tem rešili. Ta rešitev za Digizet ni primerna iz več razlogov, ki jih tu ne bomo obravnavali.

2. Tudi to vprašanje je pomembno, saj lahko izbira prevelike (ali premajhne) globine močno vpliva na hitrost izrisovanja pa tudi izločanja nevidnih ploskev. V primeru prevelike maksimalne globine se namreč lahko drevo izrodi. Če je globina prevelika, bo tudi čas za obhod drevesa ustrezno večji, kar lahko pomeni ozko grlo. Na drugi strani pa v primeru, ko je globina zelo majhna, drevo izgubi svoj pomen. V ekstremnem primeru maksimalne globine 1 bi to pomenilo, da je vsa geometrija shranjena v korenskem vozlišču, kar je isto, kot da drevesa ne bi bilo. Tako uporaba osmiškega drevesa popolnoma izgubi svoj namen. Kako pa izberemo "optimalno" vrednost? Na to vprašanje točnega odgovora ni – globino izberemo "po občutku" oz. testiramo pogon pri različnih maksimalnih globinah tako, da odkrijemo, katera meja je za dano geometrijo najustreznejša. Pri Digizetu se je na primer izkazalo, da je meja nad 10 ali pod 7 slaba. Najboljše rezultate sem dobil pri meji 8. Vendar pa je tudi ta izbira odvisna od drugih parametrov, s katerim gradimo drevo, na primer od izbire meje za maksimalno število trikotnikov v končnem vozlišču drevesa.

3. Maksimalna omejitev glede števila trikotnikov v končnem vozlišču je v Digizetu le približna. Jasno je, da lahko število otrok v listu preseže mejo, ki smo jo določili, saj lahko dosežemo maksimalno globino, še preden "porabimo" vse trikotnike v danem vozlišču. To ni problem, saj je maksimalna meja za število otrok le zaželena meja – če tega pogoja ne izpolnimo, ni nič narobe. Kriteriju za maksimalno globino namreč damo prednost. Seveda pa bi lahko dali prednost kriteriju za maksimalno število otrok in bi potem lahko presegli maksimalno globino, vendar je to praktično vedno slabša izbira, saj se lahko drevo izrodi v veliko globino, česar si ne želimo. V Digizetu torej dopuščamo, da je število trikotnikov v listu drevesa večje kot maksimalno zaželeno, globina lista pa nikoli ne preseže maksimalne meje.

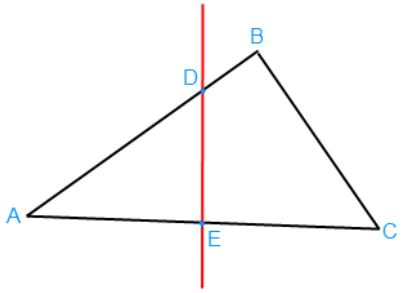
   Ostaja pa vprašanje, kakšna naj bo ta meja. Podobno kot pri prejšnjem vprašanju tudi tu ni točnega odgovora. Tako previsoka kot prenizka meja lahko zelo upočasnita delovanje pogona. V primeru prenizke meje se bo osmiško drevo preveč delilo, saj bo skušalo doseči podprostore s čim manj trikotniki. V primeru prevelike meje pa se bo delilo premalo in bo s tem namen drevesa izgubljen. Ekstremni primer je meja, ki je enaka število trikotnikov v svetu. V tem primeru bi vsi trikotniki ostali v korenskem vozlišču, saj ne bi bilo potrebe, da se to deli naprej. Podobno kot pri primeru maksimalne globine tudi tu drevo popolnoma izgubi svojo funkcijo. Mejo najlažje določimo tako, da testiramo različne meje in se odločimo za tisto, pri kateri pogon doseže najvišjo hitrost (seveda v primeru, ko skušamo pogon optimizirati za hitrost).

Razrez trikotnika

Razrez trikotnika, ki smo ga omenili kot možno rešitev problema sekanja več kot enega oktanta, bomo posebej obravnali v tem razdelku. Pokazali bomo namreč, da je ta rešitev, čeprav se morda zdi najbolj elegantna (logika za obravnavo trikotnikov se poenostavi, saj nam ni treba skrbeti za vzdrževanje referenc in zaznavanje duplikatov), v resnici slabša od tiste, ko smo referenco na trikotnik shranili v vse sekajoče oktante. Na prvi pogled se sicer izognemo podvajanju trikotnikov (večkratni izris istega trikotnika), vendar pa ustvarimo več manjših trikotnikov, ki jih moramo v celoti hraniti v pomnilniku (in ne le reference na isti objekt). Poleg tega je število teh trikotnikov večje kot število referenc na isti trikotnik.

Postopek za razrez trikotnika opravimo v treh korakih. V vsakem koraku poskusimo trikotnik prerezati z eno od osno poravnanih ravnin XY, XZ ali YZ.

Trikotnik seka ravnino tako, da je eno oglišče na eni strani ravnine, drugi dve pa na drugi. To kaže slika 2.6.



Slika 2.6: Ravnina seka trikotnik v točkah D in E.

Oglišče, ki je samo na eni strani ravnine, označimo z A, drugi dve pa B in C. Sečišče med A in B označimo z D, med A in C pa z E. Ob razrezu dobimo dva nova lika: trikotnik ADE ter štirikotnik DBCE. Zadnjega moramo triangulirati. Tu lahko izbiramo med več možnostmi, izberemo na primer: DBC in DCE. Trikotnik ADE dodamo v eno osmiško vozlišče, trikotnika DBC in DCE pa v drugo. Vidimo, da smo iz enega trikotnika dobili 3 nove. Pri hranjenju referenc na isti trikotnik pa bi hranili le 2 referenci. Čeprav so novi trikotniki manjši kot originalni, še zmeraj potrebujemo trikrat več pomnilnika za hranjenje kot za original trikotnik. Posamezen trikotnik pa lahko od osmih oktantov seka največ pet oktantov. V najslabšem primeru dobimo torej $1 \times 3 \times 3 \times 3 = 27$ trikotnikov. Če zopet primerjamo z metodo hranjenja referenc, vidimo, da bi morali uporabiti le 5 referenc, tu pa ustvarimo 27 novih trikotnikov. Vendar pa situacija ni tako črna, kot se zdi. To je ekstremen primer, do katerega pride zelo redko. Pri dani geometriji, ki je vsebovala dva nebotičnika, sestavljena iz skupaj 61272 trikotnikov, sem z razrezom dobil 174696 trikotnikov, kar je približno za faktor 3 več (ne smemo pozabiti, da nam ni treba razrezati vseh trikotnikov, zato je ta faktor malo manjši od 3). Po metodi hranjenja referenc pa sem dobil 107964 trikotnikov (torej 46692 duplikatov). To razmerje je že precej manjše.

Postavi pa se vprašanje, ali uporaba tabele za ugotavljanje duplikatov res tako zelo upočasni delovanje programa, da bi se izrez izplačal? Dejstvo namreč, da pri razrezu dobimo nove trikotnike, ki jih moramo hraniti v pomnilniku, ni tak problem, saj trikotnike hranimo v pomnilniku na grafični kartici in pri izrisovanje grafični kartici pošiljamo le indekse trikotnikov, ki naj se izrišejo (metoda VBO – “Vertex Buffer Object”). Izkaže pa se, da je ustvarjanje novih trikotnikov kljub vsemu velik problem. Bolj ko se veča geometrija, večjo kazen

plačamo za uporabo tabele duplikatov, vendar pa tudi plačamo nesorazmerno večjo kazen za indeksiranje novih trikotnikov, ki jih dobimo pri izrezu. Pri testiranju z Digizetom se je izkazalo, da se pogon upočasni tudi za faktor 2 pri uporabi razreza trikotnikov. Seveda pa to na splošno ne velja – za določene geometrije bi bil ta faktor drugačen, hitrost pa je odvisna tudi od implementacije pogona in načina, kako pogon uporablja osmiško drevo. Vendar pa se je pri konkretni implementaciji v Digizetu ta rešitev pokazala kot slaba.

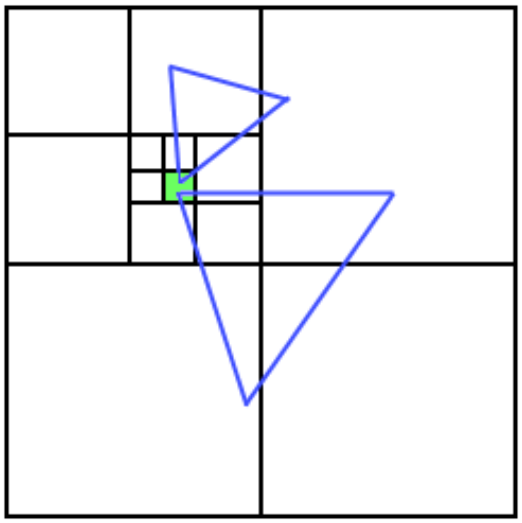
Optimizacija gradnje drevesa

V tem razdelku si bomo pogledali še en manjši popravek oz. optimizacijo pri gradnji drevesa, ki preprečuje, da bi se drevo preveč izrodilo. Eden od problemov pri gradnji, ki ga še nismo obravnavali in je značilen le za osmiško drevo (od treh obravnavanih), je ta, da se lahko število trikotnikov na vozlišče z nadaljno delitvijo ne manjša oz. se manjša zelo počasi. Smisel delitve je prav v tem, da zmanjšamo število trikotnikov na vozlišče in da tako geometrijo prostorsko razdelimo na obvladljive enote. Slika 2.7 kaže poenostavljen primer tega problema v štiriškem drevesu. Na sliki imamo dva trikotnika, za kriterij zaustavitve delitve pa smo vzeli pogoj, da je v vsakem vozlišču kvečjemu en trikotnik. V zeleno obarvanem kvadrantu nam tega ni uspelo doseči niti po štirih delitvah. V nekaterih primerih nam trikotnika ne uspe "izolirati" (t.j. zaobjeti v enem samem vozlišču) niti z neskončno globino rekurzivne delitve. Zato je rekurzijo boljše ustaviti, še preden se trikotnik prvič deli. Vendar je take primere na splošno težko zaznati. Da bi se v Digizetu izognil neučinkoviti delitvi do maksimalne globine v takih primerih, sem problem rešil z rezanjem drevesa. Algoritem deluje tako:

- Pred vsako delitvijo si najprej zapomni število objektov v vozlišču. Po delitvi pa sešteje število objektov v vseh osmih otrocih. Definiramo kvocient $branch\_coef$ kot razmerje med številom "novih" objektov (vsota objektov v otrocih) proti številu starih (število objektov v staršu).
- Če je kvocient $branch\_coef$ večji od izbranega, razveljavi zadnjo delitev (ker z njo pridobimo premalo).

Izkazalo se je, da je tak način rezanja drevesa učinkovit. Pri izrisovanju drevesa sem dobil boljše rezultate kot pri neobrezanem drevesu. Seveda bi postopek lahko tudi izpopolnili, na primer na način, da ne bi gledali le en korak v globino, temveč več korakov. Zgodi se lahko namreč, da se po nekaj korakih število objektov na vozlišče precej zmanjša (geometrija se poravna v manjših osmiških vozliščih) in je bila prvotna delitev zato smiselna.

Izbira kvocienta $branch\_coef$ je tako, kot smo videli že pri prejšnjih izbirah različnih koeficientov, zelo odvisna od geometrije in drugih dejavnikov, zato je v splošnem najboljše, da ga določimo na podlagi serije poskusov. Pri geometrije, pri kateri sem algoritem testiral, se je kvocient 1,8 izkazal za ustreznega. V splošnem kvocient izberemo v meji med 1 (kvocient ima vrednost 1 v primeru, ko noben izmed trikotnikov ni na meji med vozlišči) in 5 (to vrednost ima kvocient v primeru, ko vsi trikotniki sekajo maksimalno število možnih oktantov, to je 5) pri dupliciranju referenc trikotnikov. Pri rezanju trikotnikov bi bila zgornja meja višja, kot smo to ugotovili že v razdelku o rezanju trikotnikov.



Slika 2.7: Prostor skušamo rekurzivno razdeliti, da bi vseboval le en trikotnik v vozlišču. V nekaterih primerih to ni možno, saj se bo trikotnik zmeraj nahajal v vsaj dveh vozliščih, ne glede na globino delitve.

2.3.2 Uporaba osmiškega drevesa

Pod uporabo osmiškega drevesa mislimo predvsem na osnovne operacije nad drevesom – vstavljanje, brisanje in iskanje/dostopanje do elementov. V Digizetu je pomembna le zadnja. Elementov med tekom programa namreč ne brišemo, prav tako po izgradnji drevesa ne vstavljamo novih. Temu botruje dejstvo, da v Digizetu uporabljamo zelo omejeno dinamično geometrijo in jo raje obravnamo posebej, kot pa da bi bila del geometrije vsebovane v drevesu. Ker je del te geometrije zelo majhen, ne vpliva bistveno na hitrost izrisovanja.

O operacijah brisanja in vstavljanja pa lahko rečemo, da so načeloma zelo hitre (veliko hitrejše kot pri BSP/kd-drevesih), saj lahko drevo na enostaven način popravimo tako, da se prilagodi novi geometriji.

V Digizetu do geometrije v drevesu dostopamo na več načinov. Potrebujemo jo na primer ko:

- Potrebujemo trikotnike v okolici karakterja, za katerega želimo izvesti detekcijo trkov in odziv na trk.
- Izisujemo svet, viden skozi kamero.
- Skušamo izločiti nevidne ploskve.

Pri detekciji trkov nas tako zanimajo vsi trikotniki znotraj nekega volumna, v katerem sumimo, da je prišlo do trka. Pri premiku karakterja ta volumen izračunamo tako, da okoli karakterja postavimo obsegajočo sfero, nato pa radij te sfere povečamo za toliko, kolikor se je karakter premaknil med eno sličico in naslednjo. Iščemo torej vsa tista končna vozlišča drevesa, ki sekajo dano sfero. Pri računanju trkov med diskom in svetom pa iščemo vse tiste trikotnike, ki jih disk med eno in drugo sličico lahko zadene. Tukaj iščemo sečišče 3D daljice z drevesom. Podoben test uporabimo pri sistemu delcev, kjer posamezen delec predstavimo kot 3D točko v svetu.

Pri izrisovanju sveta je test bolj kompleksen. Potrebujemo vse trikotnike, ki sekajo piramido pogleda. Piramida je sestavljena iz šestih ploskev, zanimajo pa nas vsa končna vozlišča, ki se nahajajo znotraj piramide ali pa sekajo eno izmed ploskev.

Pri izločanju nevidnih ploskev lahko uporabimo isti test kot pri izrisovanju, lahko pa uporabimo tudi detekcijo med žarkom in drevesom, če uporabljamo na primer metodo sevanja žarkov.

Vsi ti testi pa temeljijo na rekurziji. Test zmeraj začnemo nad korenskim vozliščem in se nato rekurzivno pomikamo v globino po njegovih otrocih. Kot smo že omenili pri obravnavi BSP-dreves, se pri osmiških drevesih pomikamo zelo hitro v globino, saj je faktor vejanja štirikrat večji kot pri BSP ali kd drevesih. Zaradi te lastnosti so osmiška drevesa primerna za velike, odprte scene.

Poglavje 3

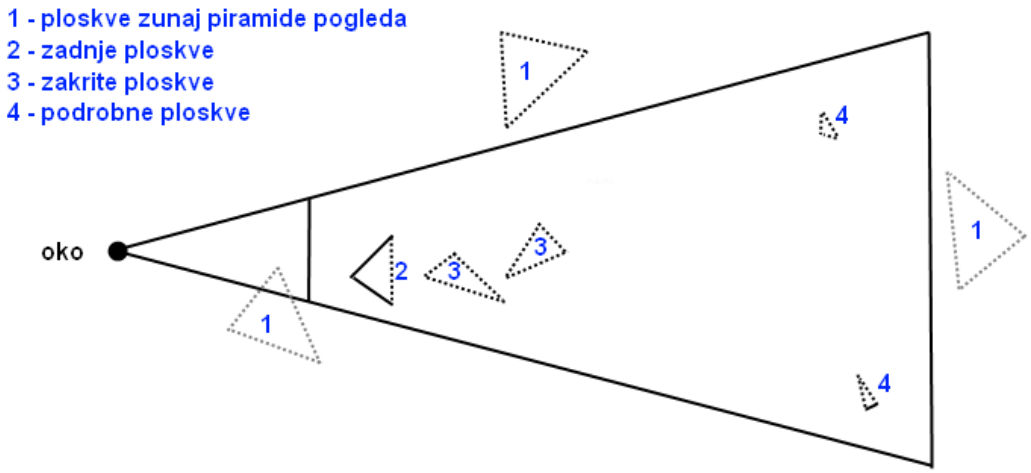
Pregled metod izločanja

Izločanje nevidnih ploskev je zelo pomembno opravilo v današnjih 3D-pogonih. Čeprav moč procesorjev in grafičnih kartic kar naprej narašča, je ne bo nikoli dovolj oz. bomo zmeraj našli načine, kako jo uporabiti. Moderne igre uporabljajo bolj in bolj podrobne scene z več milijoni trikotnikov. Pričakujemo lahko, da bo s povečevanjem hitrosti strojne opreme naraščalo tudi število uporabljenih trikotnikov v aplikacijah. Današnje grafične kartice zmorejo izrisati nekaj 100 milijonov trikotnikov na sekundo. Ker pa je večina trikotnikov, iz katerih je scena sestavljena, nevidnih, bi jih radi izločili in čas, namenjen izrisu trikotnikov, posvetili tistim, ki so vidni. Čim več trikotnikov izrišemo, tem bolj podrobna in bogata je scena. Pri izločanju torej stremimo k temu, da bi izločili čim več nevidnih trikotnikov oz. v splošnem poligonov, objektov itd. V tem poglavju si bomo pogledali glavne metode izločanja, nato pa si bomo v posebnih poglavjih bolj podrobno pogledali štiri od njih, ki sem jih implementiral tudi v Digizetu in ki so jedro te diplomske naloge.

Slika 3.1 kaže štiri tipe nevidnih ploskev. Iz tega sledijo tudi štiri skupine izločanja nevidnih ploskev:

- izločanje ploskev zunaj piramide pogleda
- izločanje zadnjih ploskev
- izločanje zakritih ploskev
- izločanje podrobnosti

V diplomski nalogi obravnavam prve tri skupine, s poudarkom na tretji, t.j. izločanju zakritih ploskev. Pri izločanju podrobnosti ("Detail culling", tudi "Contribution culling") gre za to, da objekte, ki so daleč od kamere in prispevajo le malo vidnih točk h končni sliki, izločimo. Te skupine ne obravnavam



Slika 3.1: Primer štirih skupin nevidnih ploskev.

podrobneje. Poleg omenjenih skupin metod izločanja se v 3D-pogonih uporabljajo še druge tehnike pohitritve, ki jih sicer ne prištevamo k izločanju nevidnih ploskev, vendar z njihovo pomočjo tudi izločimo del geometrije. Primer je metoda LOD ("Level of detail"). Pri tej metodi objekte, ki so dovolj daleč stran od kamere, zamenjamo z objekti, ki vsebujejo manj poligonov in so bolj preprosti. Če je objekt dovolj daleč, se razlike skorajda ne opazi. Pri tem pa moramo paziti, da je prehod med posameznimi nivoji LOD kar najbolj "gladek". Običajno v ta namen uporabljamo razne interpolacijske metode, zlivanje ipd. Primer različnih nivojev LOD prikazuje slika 3.2, slika 3.3 pa prikazuje uporabo pri različnih oddaljenostih od kamere. Razlike med posameznimi modeli skoraj ne opazimo.

V nadaljevanju si bomo na kratko ogledali nekaj bolj pogosto uporabljenih metod izločanja zakritih ploskev.

3.1 Izločanje zakritih ploskev

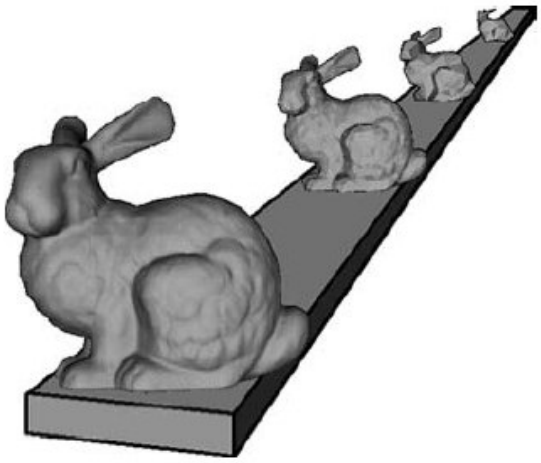
V to skupino metod izločanja sodijo vse metode, ki skušajo izločiti ploskve zakrite od drugih ploskev. Nabor vseh primitivov, ki so v celoti ali le delno vidni na zaslonu, označimo kot EVS ("Exact visible set"). Z metodami izločanja

Slika 3.2: Štirje modeli z različnim številom trikotnikov generirani iz istega modela zajca. Slika povzeta po http://habib.wikidot.com/techniques.

skušamo odstraniti čimveč primitivov, tako da je nabor primitivov, ki jih izrišemo, čimbolj podoben EVS. V praksi pa ta nabor nikoli ni enak EVS, temveč se mu le približa. Zato takemu naboru rečemo PVS ("Potentially visible set"). Čimbolj je PVS podoben EVS, temveč ploskev izločimo in tako pohitrimo izrisovanje. Seveda pa nas izračun PVS zmeraj nekaj stane. Pri izločanju skušamo doseči ugodno razmerje med časom, ki ga porabimo za izračun PVS ter časom, ki ga prihranimo z risanjem izločenih ploskev. Prizadevamo si, da je v povprečju ta razlika pozitivna. Vsakič, ko to ni tako, z izločanjem pogon upočasnimo. Vendar pa se na splošno težko ognemo vsem primerom, ko ne bi pogona vsaj malo upočasnili zaradi uporabe tehnik izločanja. V določenih primerih je na zaslonu vidna večina geometrije in zato ne moremo izločiti velikega števila ploskev. Izračunani PVS je lahko:

- Konzervativen: $\mathbf{PVS} \supseteq \mathbf{EVS}$. Pri takem PVS je slika natančno taka, kot če bi izrisali EVS, s to razliko, da izrišemo nekaj ploskev preveč.
- Približen: $\mathbf{PVS} \sim \mathbf{EVS}$. Tu toleriramo manjše napake v sliki oz. manjkajoče ploskve, ki ne prispevajo bistveno k sliki.
- Agresiven: $\mathbf{PVS} \subseteq \mathbf{EVS}$. Tu izrišemo le ploskve, ki dejansko prispevajo k sliki (spremenijo točke). Tako lahko hitro povzročimo vidne artefakte v sliki.

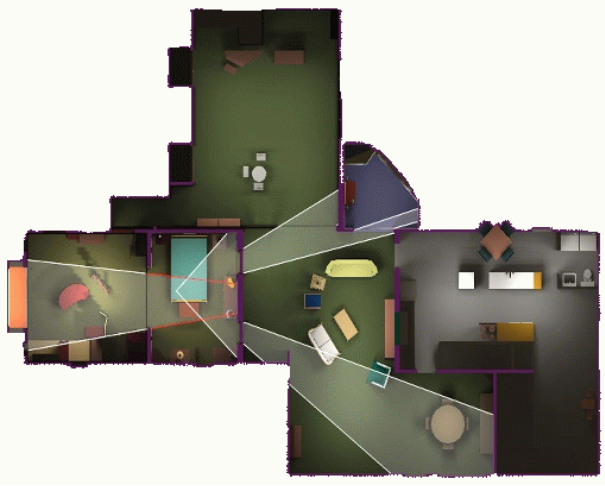
Vse metode, ki sem jih implementiral v okviru te diplomske naloge in so obravnavane v posebnih poglavjih, izračunajo konzervativen PVS. Izjema je včasih izločanje s poizvedbami zakritosti, kjer lahko definiramo prag zakritosti ploskve glede na število vidnih pik. Nekaj primerov metod izločanja zakritih ploskev:



Slika 3.3: Štirje modeli zajca s slike 3.2 postavljeni različno daleč od kamere. Slika povzeta po http://habib.wikidot.com/techniques.

- izločanje s portali ("Portal culling")
- hierarhična vidljivost ("Hierarchical Visibility Algorithm")
- sevanje žarkov ("Raycasting")
- izločanje s poizvedbami zakritosti ("Occlusion queries")
- HOM algoritem ("Hierarchical occlusion map")
- izločanje s sencami ("Shadow culling")

Izločanje s portali pride prav pri geometrijah, ki vsebujejo zgradbe. Stene (na primer v sobah) zelo dobro zakrijejo večino ostale geometrije zunaj notranjosti prostora. Če si predstavljamo, da smo postavljeni v sobo, bomo videli le to, kar je neposredno pred nami v sobi. Preostali del sveta za nas ni viden, razen skozi odprtine v sobi (okna, vrata). Te odprtine imenujemo portali, sobo pa celico. Poleg tega, da moramo odkriti vse podprostore, ki jih lahko označimo kot celice, moramo izračunati tudi vse potencialno vidne celice, ki jih lahko vidimo skozi portale v posamezni celici. V izvajalnem času pa računamo tudi vidne piramide, s katerimi določimo, kaj točno je vidno skozi portale zunaj celice. Algoritem je torej podoben algoritmu izločanja zunaj piramide pogleda. Slika 3.4 kaže primer izračuna vidnih piramid za določeno točko v sceni.

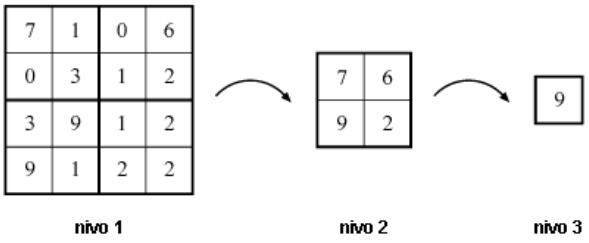


Slika 3.4: Primer vidnih piramid iz točke v centralni sobi. Slika povzeta po http://www.cs.unc.edu/∼walk/papers/luebke/portals.html.

Prednosti te metode so, da si lahko pri izračunu vidljivosti pomagamo s predprocesiranjem (določanje celic), v izvajalnem času pa izločamo predmete zunaj vidnih piramid portalov, kar računsko ni zahtevno. Na ta način lahko izločimo zelo veliko geometrije na zelo preprost način. Slabosti metode sta predvsem dve: svet mora biti zaprt, sestavljen iz celic in celice mora določiti oblikovalec sveta. Te metode torej ne moremo uporabiti na splošni sceni, ampak je primerna le za določen tip geometrije (arhitekturne scene).

Pri hierarhični vidljivosti poleg drevesne strukture (recimo osmiškega drevesa) uporabljamo tudi Z piramido. Z piramida je pravzaprav navaden Z medpomnilnik, razdeljen v več nivojev. Na prvem nivoju imamo običajen Z medpomnilnik, na naslednjem nivoju pa štirikrat manjši medpomnilnik, ki v vsaki točki vsebuje najbolj oddaljeno Z vrednost iz polja velikosti 2x2 iz prejšnjega nivoja. Slika 3.5 prikazuje rekurziven razcep Z medpomnilnika na posamezne nivoje.

Algoritem deluje takole: Obhod osmiškega drevesa opravi v vrstnem redu "najbližji najprej". Za vsako vozlišče testiraj sprednje tri stranice proti Z piramidi in če so so v celoti zakrite, vozlišče izloči. V nasprotnem primeru izriši geometrijo, ki jo vozlišče vsebuje na sliko in v Z piramido ter rekurzivno ponovi postopek na otrocih vozlišča. Testiranje z Z piramido izvedemo takole: začnemo na najbolj grobem nivoju. Od osmiškega vozlišča vzamemo tisto oglišče, ki je najbliže kameri in primerjamo z ustreno vrednostjo v Z piramidi. Če je to oglišče bolj oddaljeno kot najbolj oddaljena vrednost v Z piramidi,



Slika 3.5: Primer Z piramide. Na vsakem naslednjem nivoju obdržimo le najbolj oddaljeno vrednost iz 2x2 polja na višjem nivoju. Slika povzeta po [9].

lahko celotno vozlišče izločimo. Če ni, postopek ponovimo na višjem nivoju. Preveriti moramo vse tiste točke v Z piramidi, ki v celoti pokrijejo sprednje tri stranice vozlišča. Ta algoritem je zelo učinkovit, saj dobro zazna zakrite ploskve. Vendar pa je vzdrževanje Z piramide drago, saj jo moramo implementirati na CPE.

Metodi sevanja žarkov in poizvedbe zakritosti obravnavamo v posebnih poglavjih. Ostale algoritme, ki jih na temu mestu ne bomo obravnavali, si lahko bralec ogleda npr. v [9].

Poglavje 4

Izločanje zadnjih ploskev

Izločanje zadnjih ploskev ("back-face culling") je ena od najbolj osnovnih in preprostih tehnik izločanja. Pri izločanju zadnjih ploskev v povprečju izločimo polovico vseh ploskev, kar nam da zelo velik prihranek pri upodabljanju, saj se to izločanje izvrši še pred stopnjo upodabljanja v izrisovalniku. Dejstvo, da lahko v povprečju izločimo približno polovico vseh objektov, lahko vidmo zelo preprosto. Če bi bili na primer vsi poligoni obrnjeni proti nam, ne bi mogli izločiti nobenega. Če pa bi sedaj kamero obrnili za 180 stopinj v smeri XZ ravnine in se postavili za poligone, pa bi lahko izločili prav vse – v povprečju torej 50%. Za neko povprečno geometrijo pa lahko domnevamo, da je posamezen poligon obrnjen v naključno smer in je tako iz vseh možnih postavitev kamer mogoče v povprečju izločiti 50% vseh poligonov.

Pomembna predpostavka pri izločanju zadnjih ploskev je, da so vsi objekti v našem 3D svetu "zaključeni" oz. zaprti. "Notranjosti" objektov torej ne moremo videti. Predstavljajmo si zgradbo – če zgradbo gledamo od zunaj, njene notranjosti ne moremo videti (razen skozi odprta vrata ali okno). Podobno ne vidimo notranjosti omare, stola ali drugih "trdnih" (angleško "solid") objektov – same notranjosti noge stola ne moremo videti, ker se nikoli ne moremo postaviti v njeno notranjost. Zato vse trikotnike, ki sestavljajo stol in so obrnjeni proč od nas, nikoli ne izrišemo. Kaj pa storimo v primeru, ko ta predpostavka ni izpolnjena? Izkaže se, da lahko to predpostavko vedno uresničimo, in sicer na zelo preprost način: notranjost objektov, ki so votli in v katere se lahko postavimo, "oblečemo" tudi od znotraj – torej samo preslikamo trikotnike, ki sestavljajo njeno zunanjost, tako, da jih obrnemo (spremenimo orientacijo). Edini problem je ta, da imajo določeni objekti dvojno "steno", drugi pa ne, vendar se da ta postopek "oblačenja" posameznih objektov avtomatizirati (pameten program za 3D-modeliranje lahko sam ugotovi, katerih

poligonov se nikoli ne vidi). Ta problem je torej le "lepotne" narave (v smislu načrtovanja sveta – uporabnik ne bo opazil nikakršnih vizualnih razlik). Po drugi strani pa velja: če imamo takih "odprtih" objektov veliko, imamo tudi podvojenih poligonov sorazmerno veliko. V tem primeru si lahko mislimo, da bi bilo enako dobro uporabiti običajno obravnavanje poligonov (izrisovanje obeh strani), tako da bi se s tem izognili dvojnemu "oblačenju" objektov. S tem bi lahko bistveno znižali število poligonov v svetu. V kakšnem primeru pa bi se to dejansko splačalo? V primeru, ko bi izločanje zadnjih ploskev implementirali na nivoju CPE, torej v programskem ("software") načinu. S tem bi se izognili pošiljanju velike količine geometrije na grafično kartico (v povprečju bi jo zmanjšali za polovico). Slabost tega pristopa pa je, da v praksi ne moremo doseči take pohitritve s programskim rezanjem zadnjih ploskev, kot jo lahko dosežemo na GPE. Na CPE moramo za vsak poligon izračunati skalarni produkt njegove normale z vektorjem pogleda, kar predstavlja več zaporednih operacij, ki jih GPE lahko vzporedno izvrši. Druga slabost tega pristopa pa je, da s tem ne bi mogli bistveno pohitrili izrisovanja, saj dandanes GPEji zelo hitro in učinkovito projicirajo 3D-geometrijo v 2D "frame buffer", poleg tega pa GPE ob uporabi strojne Z-buffer tehnike poskrbi za to, da ne pride do pretiranega prerisovanja slikovnih točk, kar še dodatno zmanjša uporabnost programskega načina izločanja zadnjih ploskev. Še zadnji razlog je ta, da je primerov, ko bi dejansko potrebovali izrisovanje obeh strani poligona, zelo malo. V večini primerov, ko dejansko potrebujemo tanke "stene" objektov, so te stene na notranji strani drugačne barve oz. so obljepljene z drugačno teksturo kot na zunanji strani, tako da tudi v tem primeru potrebujemo dvojno steno. Dejansko pa v večini 3D-svetov prevladujejo trdni ("solid") objekti, kar je še dodaten razlog za uporabo strojnega izločanja zadnjih ploskev.

V Digizetu sem poskusil implementirati oba načina rezanja zadnjih ploskev – tako programskega kot strojnega.

4.1 Strojna implementacija

Strojna implementacija pomeni uporabo ustrezne zastavice v OpenGL, ki nato poskrbi za vse, kar je potrebno, da grafična kartica uporabi to tehniko rezanja. To dosežemo s funkcijo:

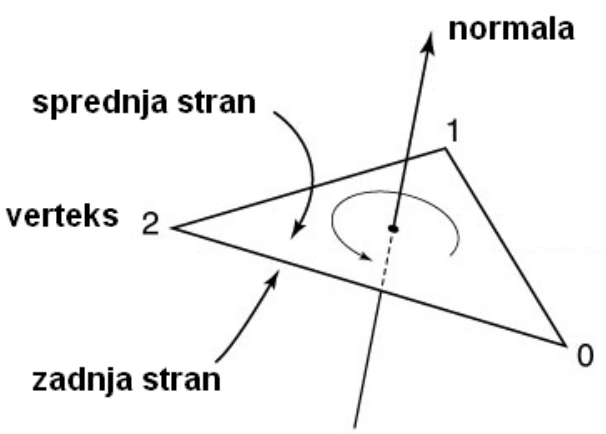
```
glEnable(GL11.GL_CULL_FACE);
```

Poleg tega lahko nastavimo tudi, katero stran poligonov bo rezal GPE – spred-

njo, zadnjo ali pa obe. To storimo z naslednjo funkcijo:

```
glCullFace(int mode);
```

Pri čemer je `mode` ena od treh možnih vrednosti: `GL_FRONT_AND_BACK`, `GL_FRONT`, `GL_BACK`. Orientacija je pomembna – pri vseh poligonih moramo uporabljati enako orientacijo, sicer algoritem ne bo deloval pravilno. Običajno imamo za sprednjo stran tisto stran poligona, pri kateri so robne točke podane v obratni smeri urnega kazalca, kot to kaže slika 4.1. Če uporabimo nasprotno smer, lahko to nastavimo z uporabo funkcije `glFrontFace(int mode)`, kjer je `mode` ena od dveh možnih vrednosti: `GL_CW` in `GL_CCW` (CW – smer urnega kazalca, CCW – obratna smer urnega kazalca).

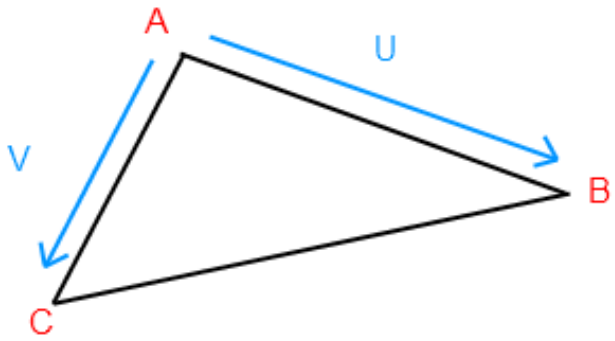


Slika 4.1: Sprednja in zadnja stran trikotnika glede na smer označevanja vozlišč.

4.2 Programska implementacija

V Digizetu programsko preverjamo trikotnike, medtem ko rekurzivno pregledujemo osmiško drevo in zbiramo vse trikotnike, ki bodo najverjetneje vidni in jih bomo v naslednjem koraku poslali na grafično kartico. Pred tem pa jih še uredimo v teksturna vedra, jih uredimo po indeksih ter grupiramo tako, da s čim manj klicev OpenGL pošljemo celotno geometrijo izrisovalniku. Za ugotavljanje zadnje strani trikotnika potrebujemo normalo na trikotnik ter vektor pogleda. Vse normale izračunamo že vnaprej, tako da nam jih ni treba

računati sproti (to bi bil računsko prevelik zalogaj). Normalo izračunamo po naslednjem postopku:



Slika 4.2: Vektorja U in V vzdolž trikotnika.

$$N = (B - A) \times (C - B)$$

oziroma če zapišemo drugače:

$$N.x = U.y \times V.z - U.z \times V.y$$
$$N.y = U.z \times V.x - U.x \times V.z$$
$$N.z = U.x \times V.y - U.y \times V.x$$

kjer je N vektor normale, U in V pa vektorja v smeri $A \to B$ oz. $A \to C$ (glej sliko 4.2). Takšna normala še ni normalizirana (nima dolžine 1). Zato je potreben še en korak:

$$n = \sqrt{(N.x)^2 + (N.y)^2 + (N.z)^2}$$
$$\widehat{N}.x = \frac{N.x}{n}$$
$$\widehat{N}.y = \frac{N.y}{n}$$
$$\widehat{N}.z = \frac{N.z}{n}$$

Vektor pogleda je definiran kot vektor od točke pogleda (kamere) do poljubne točke na trikotniku. Da ugotovimo, ali vidimo trikotnik z zadnje strani, moramo le še pogledati predznak skalarnega produkta med vektorjem pogleda

ter normalo trikotnika:

$S = (\overrightarrow{oko} - \vec{A}) \cdot \vec{N})$

kjer je $\overrightarrow{oko}$ vektor pogleda, $\vec{A}$ je poljubno oglišče trikotnika, $\vec{N}$ pa normala trikotnika. Če je $S > 0$, potem trikotnik izločimo, saj je proti nam obrnjen z zadnjo stranjo.

4.3 Meritve

Pri podani statični geometriji se je izkazalo, da je bilo strojno izločanje zadnjih strani precej bolj učinkovito, in sicer v povprečju od 10 do 20%. Da pa bi lažje odkril povezavo med pohitritvijo oz. upočasnitvijo in programskim izločanjem, sem isti test ponovil pri "simuliranem" programskem izločanju. To pomeni, da sem programsko kodo prepisal tako, da sem izločil le tisti del, ki dejansko izloča trikotnike, vse ostalo pa je ostalo enako (torej računanje skalarnega produkta itd.). Tako se je izrisovalniku poslala vsa geometrija in ne le vidni trikotniki. Iz meritev lahko sklepamo, da je ozko grlo pravzaprav pošiljanje geometrije na grafično kartico in ne strojno izločanje. Pri vklopu/izklopu strojnega izločanja in izklopljenem programskem izločanju razlike v pohitritvi praktično ni bilo. Najbolj pomembna ugotovitev je torej, da programsko preračunavanje vidnosti trikotnikov najbolj upočasni celoten postopek risanja.

Strojno	Programsko	tri./s	FPS
da	ne	8,6M	204
ne	da	7,1M	167
ne	simulirano	6,6M	155
da	simulirano	6,6M	155
ne	ne	8,6M	203
ne	prerok	9,0M	209

Tabela 4.1: Rezultati testa pri geometriji level3.obj s shranjeno pozicijo kamere. Pri tem sem vzel maksimalne vrednosti na intervalu 5 sekund. V tabeli podajam dve različni meri: FPS ("frames per second", število sličic na sekundo) ter število trikotnikov poslanih izrisovalniku na sekundo.

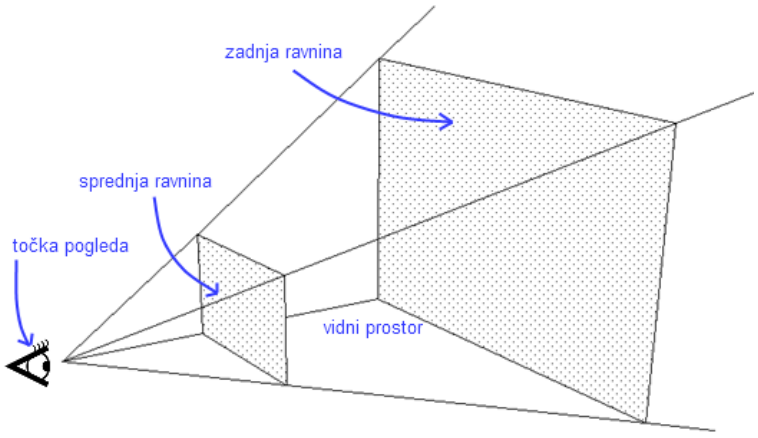
Ob rezultatih meritve iz tabele 4.1 se nam postavi vprašanje, kakšno maksimalno pohitritev bi lahko dosegli, če bi programsko izločanje uspeli še pospe-

šiti? Iz rezultatov namreč vidimo, da lahko z zgodnjim rezanjem (preden pošljemo geometrijo izrisovalniku) pridobimo na hitrosti. V ta namen sem razvil še en test: vidnost trikotnikov sem izračunal vnaprej in rezultate shranil v tabelo, algoritem za programsko izločanje pa popravil tako, da je med samim izvajanjem programa namesto preračunavanja, ali je trikotnik viden ali ne, rezultat prebral iz vnaprej pripravljene tabele in se na podlagi teh podatkov odločil, ali je objekt viden ali ne oz. ali ga bo poslal naprej izrisovalniku. Rezultat te meritve sem v tabeli označil kot programski način izločanja "prerok". Vidimo, da smo tako dosegli še dodatno pohitritev (približno 5% pri dani geometriji), vendar pa v resničnosti ne moremo pričakovati, da bomo računanje vidnosti trikotnika pohitrili do take mere. Teoretično lahko sicer časovno kompleksnost algoritma spravimo pod linearno, na primer na način, kot to predlaga Johannsen et al. [6] in sicer z metodo grupiranja ("Clustered Backface Culling"). Ideja metode v grobem je, da podobno orientirane poligone, ki so blizu skupaj, naenkrat označimo za obrnjene stran od nas. Tako nam ni treba preverjati vsakega poligona posebej, ampak celotno gručo poligonov. Preprost primer je kocka – pri kocki vemo, da naenkrat vidimo kvečjemu tri stranice. Pri tem točno vemo, katerih treh ne vidimo (preostalih treh). Za vsak poljuben nabor treh stranic lahko takoj ugotovimo, katerih stranic ne vidimo, brez preverjanja vsake stranice posebej. Pri bolj kompleksnih modelih je lahko prihranek na ta način zelo velik.

Poglavje 5

Izločanje predmetov zunaj piramide pogleda

Ena od osnovnih tehnik izločanja, ki jo mora vsebovati vsak resen 3D-pogon, je tehnika izločanja predmedov zunaj piramide pogleda ("view frustum" ali tudi "viewing frustum"). Pri perspektivni projekciji definiramo piramido pogleda kot prisekano piramido, ki definira prostor pogleda v 3D-svetu. Pri drugih projekcijah vidni prostor nima nujno oblike prisekane piramide. Tak primer je vzporedna projekcija, kjer je vidni prostor v obliki kvadra.



Slika 5.1: Piramida pogleda, ki definira vidni prostor, je definirana s šestimi ravninami.

Takoj na začetku pojasnimo pojem piramide pogleda: v angleščini "frustum" pomeni na vrhu prisekan stožec ali piramido. Podobnega izraza v slovenščini nimamo, vendar pa se v literaturi uporablja pojem stožca pogleda, ki

morda ni najbolj ustrezen, saj dejansko ne gre za stožec, ampak za prisekano piramido. Pri očesu pa vidni prostor lahko opišemo s stožcem. Ker pa v računalništvu vedno uporabljamo piramido, saj je ta povezana s projekcijo kamere, bom od tu naprej uporabljal prevod "piramida pogleda".

S slike 5.1 vidimo, da piramida pogleda omejuje vidni prostor s šestimi ravninami: sprednjo in zadnjo ter štirimi stranskimi (zgornjo, spodnjo, desno in levo). Sprednja ravnina je pravzaprav projekcijska ravnina – kvader, ki ga skupaj s to ravnino definirajo stranske ravnine, je 2D-slika, ki jo izrišemo na zaslon. Zadnja ravnina pa omejuje daljavo pogleda. Dlje od točke pogleda kot je, dlje v daljavo vidimo. Piramida pogleda je torej popolnoma definirana z opisanimi šestimi ravninami. Oddaljenost točke pogleda (gorišča) od sprednje ravnine (goriščna razdalja) pa obenem definira vidni kot ("angle of view". Še bolj pogosto se uporablja izraz "field of view", ki pa pomeni nekaj drugega in je praviloma izražen z razdaljo in ne kotom). Krajša kot je ta razdalja, večji je vidni kot. Ta kot pa neposredno vpliva tudi na število izrisanih predmetov – večji kot je, večji vidni prostor oklepa piramida in s tem je vidnih več predmetov. Človek ima vidni kot približno 200 stopinj horizontalno (binokularno. Posamezno oko pa okoli 160 stopinj) ter 135 stopinj vertikalno (vendar najbolje vidi le v osrednjem delu tega razpona – 25% možganske skorje je namenjene procesiranju osrednjih 5 stopinj vidnega kota) [7]. Pri igrah običajno definiramo vertikalni vidni kot 45 stopinj (včasih je bila ta vrednost bolj pomembna zaradi poenostavitve enačb in s tem hitrejšega trigonometričnega preračunavanja, danes pa temu ni več tako in uporabljajo se poljubni koti). Manjši kot je ta kot, bolj pride do izraza učinek povečave in s tem izgube občutka za globino oz. perspektivno projekcijo, vendar pa se tudi zmanjša prostornina vidnega prostora in s tem število predmetov, ki jih lahko vidimo naenkrat (s tem dosežemo večjo pohitritev). Večji kot je ta kot, bolj pride do izraza učinek ribjega očesa ("fish eye") in s tem popačenje perspektivne projekcije.
V OpenGL se za definiranje piramide pogleda uporablja funkcija `glFrustum`:

```
void glFrustum(GLdouble left, GLdouble right, GLdouble bottom, GLdouble
top, GLdouble near, GLdouble far)
```

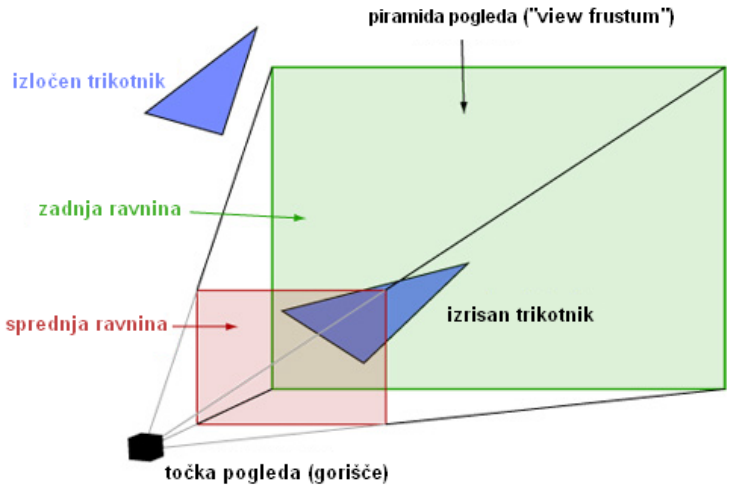
Posamezni argumenti pomenijo odmik na oseh x, y in z, pri čemer je gorišče postavljeno v točko (0,0,0). Vendar pa se v praksi bolj pogosto uporablja funkcija `gluPerspective`:

```
void gluPerspective(GLdouble fovy, GLdouble aspect, GLdouble near,
```

```
GLdouble far)
```

Parameter `fovy` pomeni vertikalni vidni kot, horizontalni kot pa se izračuna iz razmerja med višino in širino zaslona, ki ga podamo kot parameter `aspect`. `near` in `far` parametra označujeta koordinati sprednje in zadnje ravnine in s tem tudi ravnini, ob katerih grafična kartica samodejno reže geometrične primitive.

Izločanje s pomočja piramide pogleda izkorišča dejstvo, da vidimo le objekte, ki so znotraj piramide in pa delno vidne objekte, ki sekajo eno ali več ravnin, ki piramido določajo. Vse, kar je zunaj piramide pogleda, lahko izločimo iz množice primitivov, ki jih pošljemo na grafično kartico za izris (slika 5.2). Izločanje torej delamo programsko, v prostoru objektov ("object space") in ne na grafični kartici. Testiranje, ali je nek primitiv (običajno trikotnik) znotraj piramide pogleda, ni drag (v smislu časa potrebnega za izračun testa) glede na druge, bolj kompleksne tehnike izločanja. Je pa vseeno bolj kompleksen od tehnike izločanja zadnjih ploskev, za katero smo rekli, da v povprečju izloči polovico vseh vidnih primitivov in je najbolj preprosta tehnika izločanja. Pri tehniki izločanja piramide pogleda pa odstranimo še precej več primitivov, v povprečju več kot s katerokoli drugo tehniko izločanja. Če si predstavljamo prostornino piramide pogleda proti prostornini celotnega trodimenzionalnega sveta, ki ga izrisujemo, si lahko mislimo, da piramida pogleda predstavlja le majhen del celotnega prostora.



Slika 5.2: Primer izločenega in izrisanega trikotnika.

Kljub temu, da lahko s to tehniko izločanja izločimo pretežni del primitivov, pa praviloma nikoli ne izvajamo izločanja na nivoju posameznih primitivov, saj bi to vzelo preveč časa. Čeprav je posamezen test, ali je primitiv znotraj piramide, relativno poceni, pa je povsem nepraktično testirati vse primitive v svetu, ki jih vsebuje na primer nekaj milijonov. Zato ta test izvajamo na ravni obsegajočih teles ("bounding volumes"). Temu postopku pravimo tudi hierarhično izločanje predmetov izven piramide pogleda ("hierarchical view frustum culling").

5.1 Hierarhično izločanje predmetov izven piramide pogleda

Če uspemo za neko gručo objektov s preprostim testom ugotoviti, da leži zunaj vidnega prostora, lahko izločimo celotno gručo, pri čemer nam ni treba testirati posameznih poligonov, ali ležijo znotraj piramide pogleda. Velja tudi obratno: če neka gruča leži v celoti znotraj piramide pogleda, vemo, da vsi njeni elementi prav tako ležijo znotraj piramide in nam jih ni treba posebej testirati. To je tudi ideja hierarhičnega izločanja. Objekte v svetu imamo namreč praviloma urejene v neko hierarhično strukturo (graf scene), na primer BSP-drevo, osmiško drevo, kd-drevo, BVH ("bounding volume hierarchy") ipd. Če za neko vozlišče v drevesu ugotovimo, da je v celoti v piramidi pogleda ali zunaj nje, lahko to ugotovitev rekurzivno prenesemo na vse otroke tega vozlišča in s tem prihranimo veliko testiranja. Tudi če je neko vozlišče le delno prisotno v vidnem prostoru, ga lahko rekurzivno testiramo in pridemo do vozlišč, ki so zopet popolnoma definirana v smislu pripadnosti vidnemu prostoru. Postopek pa začnemo izvajati na korenskem vozlišču, ki je označeno, kot da delno pripada vidnemu prostoru. Liste drevesa, ki sekajo vsaj eno od ravnin (torej le delno pripadajo vidnemu prostoru), običajno izrišemo v celoti in ne testiramo posameznih primitivov, ki jih vsebujejo.

Pomembna opazka pri hierarhičnem izločanju je ta, da stvar sicer dobro deluje pri statični geometriji, pri dinamični pa se pojavijo problemi. Dinamična geometrija se namreč spreminja (recimo spreminja lego, obliko, ...), kar pomeni, da so lahko primitivi obsegajočih teles v enem trenutku prisotni v enem vozlišču grafa scene, v drugem pa v drugem. Modificiranje dreves (na primer BSP, osmiško) pa je lahko zelo drago in se mu izogibamo. BSP-drevesa, na primer, so še posebej draga glede konstruiranja in jih zmeraj zgradimo vnaprej, v realnem času pa jih ne spreminjamo. Na drugi strani pa ta problem preveč ne vpliva na BVH in je na splošno bolj fleksibilna struktura. Vendar pa so BSP

in osmiška drevesa bolj učinkovita (v smislu hitrost) pri statični geometriji. Problem dinamične geometrije lahko rešimo na primer na dva načina:

- uporabimo ločeno drevo, kjer shranjujemo le dinamično geometrijo. To drevo je neodvisno od drevesa, kjer hranimo statično geometrijo in je zato lahko tudi drugega tipa – recimo BVH, pri statični geometriji pa uporabimo BSP-drevo.

- uporabimo hibridno metodo: primitive v vozliščlih grafa scene obdamo z obsegajočimi telesi, ki se jim čim tesneje prilegajo. Primer: osmiško drevo, ki ima primitive v vozliščih povezane v obsegajoča drevesa.

5.2 Implementacija v Digizetu

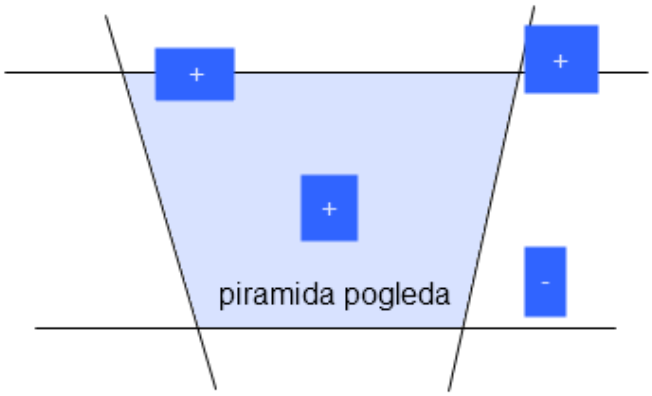
V Digizetu sem kot podatkovno strukturo za implementacijo drevesa uporabil osmiško drevo, ki ga uporabljam tudi pri hierarhičnem izločanju po metodi piramide pogleda. Osnovni test je preprost: preveriti moramo, ali kocka, ki predstavlja volumen osmiškega vozlišča, leži znotraj piramide pogleda. Vsako osmiško vozlišče ima namreč obliko kocke, pri čemer je vsako vozlišče, če ima otroke, sestavljeno iz osmih kock, kot kaže slika 2.3.
Test v grobem deluje takole:

1. Za vseh šest ravnin piramide pogleda preveri, ali kocka seka ravnino piramide ali pa leži na eni od obeh strani ravnine – na pozitivni ali negativni. Negativno stran ravnine definiramo v smeri njene normale, to je proti središču piramide, v vidni prostor, pozitivno pa obratno.

2. Takoj ko najdemo eno ravnino, pri kateri leži kocka v celoti na pozitivni strani, test končamo – celotna kocka leži zunaj piramide pogleda.

3. Če smo pri vseh šestih testih dobili za rezultat, da kocka v celoti leži na negativni strani ravnine, potem je rezultat testa ta, da kocka v celoti leži znotraj piramide pogleda. Če pa smo pri enem ali več testih ugotovili sekanje z ravnino, potem je kocka le deloma znotraj piramide pogleda. V tem zadnjem primeru je možna izjema: lahko je zunaj piramide pogleda, in sicer tako, da seka 3 ravnine, pri ostalih treh pa leži na negativni strani. Tak poenostavljen primer v 2D-situaciji prikazuje slika 5.3. V Digizetu tega problema ne obravnavam, ker je tak primer zelo redek in posebna obravnava take situacije ne bi bistveno pohitrila pogona, temveč bi vnesla še več dodatnega računanja, ki ga potrebujemo za zaznavanje

take situacije. Ena preprosta rešitev pa je ta, da bi v primeru, ko kocka seka točno tri ravnine, pri ostalih pa je na negativni strani, preverili, ali oglišče, kjer se te tri ravnine sekajo, leži znotraj kocke. Če ne leži, potem se kocka v celoti nahaja zunaj piramide pogleda in jo lahko izločimo.



Slika 5.3: 2D primer testiranja pripadnosti pravokotnikov vidnemu prostoru. Pravokotnik v desnem zgornjem kotu je klasificiran kot "pripada", čeprav je dejansko zunaj vidnega prostora.

Pri tem algoritmu uporabljamo test med kocko in ravnino, ki deluje takole:

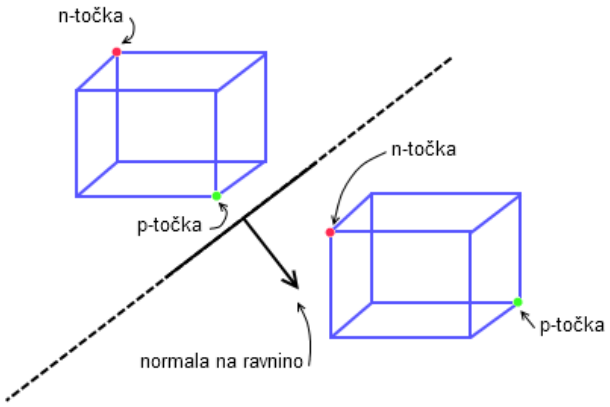
1. Izberi dve točki p in n ("pozitivna" in "negativna") izmed osmih oglišč kocke, za kateri velja: točka p je točka, ki je najbolj oddaljena od ravnine v smeri normale, točka n pa je najbolj oddaljena v obratni smeri normale. To prikazuje slika 5.4.

2. Izračunaj oddaljenost točke n do ravnine. Če je negativna, leži kocka v celoti na pozitivni strani ravnine in test je zaključen. V nasprotnem primeru pojdi na naslednji korak.

3. Izračunaj oddaljenost točke p do ravnine. Če je negativna, potem kocka seka ravnino, v obratnem primeru pa v celoti leži na njeni negativni strani.

Računanje oddaljenosti točke do ravnine oz. ugotavljanje, na kateri strani ravnine leži, lahko opravimo tudi s skalarnim produktom, ki je približno enako

kompleksna operacija. Za posamezen test med kocko in ravnino v najslabšem primeru potrebujemo torej 2 skalarna produkta. V primeru, ko je kocka v celoti na pozitivni strani ravnine, pa le enega.



Slika 5.4: p in n točki testa kocke z ravnino. Točka p je tisto oglišče kocke, ki je od ravnine najbolj oddaljeno v smeri normale, točka n pa je najbolj oddaljeno oglišče kocke od ravnine v obratni smeri normale.

Iz algoritma je razvidno, da poln test potrebuje 12 skalarnih produktov. Te operacije niso posebno drage, vendar moramo ob kompleksni geometriji testirati veliko število osmiških vozlišč, kar zahteva svoje. Zato sem v pogonu implementiral dve optimizaciji, kot ju predlagata avtorja v članku [11] in izmeril, koliko dejansko pospešita preračunavanje.

Prva optimizacija izkorišča dejstvo, da lahko otroci vozlišča sekajo kvečjemu tiste ravnine, ki jih seka tudi starš. Če to informacijo prenesemo s staršev na otroke, lahko precej zmanjšamo število testov kocke z ravnino. V Digizetu to informacijo rekurzivno podajamo otrokom (drevo zmeraj preiskujemo rekurzivno) v obliki šestih bitov – vsak bit pove, ali starš seka eno od šestih ravnin. V primeru, da je starš v celoti znotraj piramide pogleda, otrok ne testiramo, ampak jih samo izrišemo. Pri obravnavi otrok tako testiram samo ravnine, ki imajo nastavljen ustrezen bit. Drugih ravnin vozlišče zagotovo ne seka. Pri tem testu pa se lahko zgodi, da vozlišče seka kakšno ravnino manj kot njen starš (ali pa sploh nobene). V tem primeru ustrezno popravljene bite rekurzivno posredujemo otrokom tega vozlišča. Na začetku izvajanja algoritma za korensko vozlišče določimo, da seka vseh šest ravnin.

Druga optimizacija pa izkorišča lokalnost med zaporednimi sličicami ("frame-to-frame coherency") – običajno se med posameznimi sličicami kamera bistveno ne premakne, zato izrisujemo približno podobno geometrijo. Kar je bilo zunaj piramide pogleda v sličici n, bo verjetno zunaj nje tudi v sličici n+1. Ugotovili pa smo že, da za ugotavljanje tega, ali je kocka zunaj piramide, potrebujemo le en test z ravnino – in sicer s tisto, pri kateri kocka leži na njeni pozitivni strani (takih ravnin je lahko tudi več). To informacijo (indeks ravnine, pri kateri je test pokazal, da kocka leži na njeni pozitivni strani) zapišem v osmiško vozlišče in jo nato pri računanju naslednje sličice uporabim tako, da najprej testiram to ravnino. V večini primerov bo test tudi tokrat pokazal, da je kocka na pozitivni strani ravnine in tako ni potrebe po testiranju ostalih petih ravnin. Če pa temu ni tako, izvedem še ostale teste in ustrezno popravim to informacijo.

Učinkovitost teh dveh optimizacij sem testiral pri opisani geometriji:

1. Na zaslonu vidnih (ali delno vidnih) 15.036 trikotnikov. Vseh trikotnikov v svetu: 102.392. Skupaj obiskanih 5201 osmiških vozlišč.
2. Od vseh obiskanih vozlišč je bilo:
   (a) 439 takih, pri katerih je test pokazal, da so v celoti znotraj piramide pogleda (pri tem ne štejemo otrok teh vozlišč, saj nam otrok ni bilo treba testirati)
   (b) 690 takih, pri katerih je test pokazal, da so v celoti zunaj piramide pogleda (otrok tudi tu nismo preverjali)
   (c) 1160 takih, ki so bili delno vidni

Rezultati meritev so podani v tabeli 5.1. Pri tem sem beležil število testov kocke z ravnino (število testov med kocko in piramido pogleda ostaja nespremenjeno). Vsak tak test "stane" od 1 do 2 skalarna produkta, kot je že opisano.

Kot vidimo iz tabele, je prva optimizacija precej zmanjšala število potrebnih testov – približno za faktor 4. To lahko preprosto razložimo: od vseh testov je bilo 1160 takšnih, pri katerih je test pokazal, da so kocke delno vidne. Pri teh testih je optimizacija prišla v polnosti do veljave. Predpostavljamo namreč, da večina kock seka le eno ravnino. Takih, ki bi sekali dve, je zagotovo manj, še manj pa je teh, ki bi sekale vse tri ravnine. V primeru, ko starš seka le eno ravnino, je potrebno pri vseh osmih otrocih testirati le to ravnino namesto šestih. V najboljšem primeru torej število testov z ravninami zmanjšamo s 6 na 1, kar velja za približno 40% vseh testov s piramido pogleda. Pri 690 primerih, ko je osmiško vozlišče v celoti zunaj piramide pogleda, lahko

opt1	opt2	št. testov
ne	ne	10839
ne	da	10284
da	ne	2766
da	da	2612

Tabela 5.1: Rezultati testiranja pohitritve dveh različnih optimizacij opt1 in opt2 hierarhičnega izločanja predmetov zunaj piramide pogleda. Test se je izvajal pri dveh zaporednih sličicah, kjer se lega in orientacija kamere nista spreminjali – vidna geometrija je torej enaka v obeh sličicah.

naredimo podobno predpostavko: starši teh vozlišč so sekali v povprečju eno ravnino, zato je zadoščal en sam test, ki je pokazal, da je kocka v celoti zunaj piramide. Poleg tega je bilo tudi v primeru, ko je starš sekal več ravnin, potrebnih manj testov, kot je bilo vseh ravnin, ki jih je sekal starš: v primeru, ko je kocka v celoti zunaj piramide, zadostuje že en sam test, da to dokažemo (ni pa nujno, da se nam posreči, da je to ravnina, ki jo najprej preverimo). Pri preostalih 439 obiskanih vozliščih, ki so znotraj piramide pogleda, lahko spet uporabimo podobno logiko – ta vozlišča so prav tako imela starše, ki so sekali le posamezne ravnine, pretežno pa le eno. Spodnja meja, ki je namenjena za optimistično oceno (tu prezremo dejstvo, da smo korensko vozlišče označili tako, da seka vseh 6 ravnin), je torej:

$$1160 \cdot 1 + 690 \cdot 1 + 439 \cdot 1 = 2289$$

kar pa je zelo blizu rezultatom poskusa. S prvo optimizacijo smo se dejansko precej približali teoretični meji, do katere je še mogoče optimizirati algoritem. Zato nas dejstvo, da se druga optimizacija ni odrezala tako dobro, ne preseneti.

Pri drugi optimizaciji lahko optimiziramo le tista vozlišča, ki so bila v prejšnji sličici označena kot popolnoma zunaj piramide pogleda. Takih vozlišč je 690. Največ, kar lahko v tem primeru pridobimo, je torej $690 \cdot 5 = 3450$. Dejansko v povprečju porabimo manj kot 3 teste za situacijo, ko je kocka zunaj piramide – namreč če bi rekli, da take kocke ležijo v povprečju zunaj natanko ene ravnine, bi s testom tako ravnino zadeli v 3,5 poskusih. Vendar pa pri tem ne upoštevamo, da je veliko kock na pozitivni strani dveh ali celo treh ravnin. S tem bi se število poskusov, v katerih "zadenemo" takšno ravnino, precej zmanjšalo. Iz tabele razberemo, da smo prihranili 555 testov (brez vklopljene prve optimizacije), kar pomeni, da smo v povprečju število testov z ravnino na en test s piramido zmanjšali za manj kot 1. Tudi to se ujema s prejšnjim

razmislekom: najmanjše možno število poskusov na test je 1 in v naši meritvi smo s to optimizacijo tudi zagotovo dosegli to mejo, saj se geometrija med posameznimi sličicami ni spreminjala. Iz tega lahko izračunamo, da je brez optimizacije potrebnega v povprečju 1,8 poskusa, kar se ujema s prejšnjimi ugotovitvami: rekli smo, da bi v primeru, ko bi kocka ležala zunaj le ene ravnine, povprečno število poskusov padlo na 3,5, če pa upoštevamo dejstvo, da veliko kock leži zunaj dveh ali treh ravnin, pa bi to število lahko še razpolovili – kar nam da številko blizu 1,8.

Iz meritev smo ugotovili, da se število testov med kocko in ravnino precej zmanjša (približno za faktor 4). Postavlja pa se vprašanje, koliko to dejansko vpliva na hitrost izrisovanja? V opisani meritvi sem izmeril, da se je število sličic v povprečju povečalo za 1,2 odstotka, prav tako število izrisanih trikotnikov na sekundo. Ta podatek govori o tem, da je testiranje vidnosti osmiških vozlišč s piramido pogleda dejansko zelo "poceni" operacija in da močna pohitritev že tako hitre operacije ne vpliva bistveno na hitrost izvajanja programa. Včasih deluje celo nasprotno: zaradi večje kompleksnosti implementacije se je v takih primerih bolje odločiti za neuporabo optimizacije, saj je manjša kompleknost kode tudi zelo pomemben faktor pri implementaciji pogona.

Poglavje 6

Izločanje z metodo sevanja žarkov (Raycasting)

Sevanje žarkov je metoda, ki ni namenjena izključno izločanju nevidnih ploskev, temveč se v splošnem uporablja predvsem pri izrisovanje sveta. Njen "stranski učinek" je ta, da lahko z njo na preprost način izračunamo potencialno vidno geometrijo (PVS).

Metoda sevanja žarkov je podobna metodi sledenja žarkov ("raytracing"). V začetku sta se oba pojma uporabljalo za isto metodo, pozneje pa je prišlo do razlik. Pri metodi sevanja žarkov skušamo izrisati 2D-sliko tako, da pošljemo žarke skozi vsako točko ("pixel") na zaslonu in gledamo, kaj zadene žarek. Barvo točke izračunamo iz lastnosti površine, ki jo je zadel žarek. Pri metodi sledenja žarkom pa ne vzamemo le te površine, temveč rekurzivno sledimo odboju žarka in barvo točke določimo iz lastnosti vseh površin, od katerih se je odbil žarek. Sevanje žarkov je pravzaprav poenostavljena metoda sledenja žarkom, pri čemer sevamo le primarne žarke in ne sledimo odbojem. Sledenje žarkom uporabljamo na primer za izris fotorealističnih slik. Vendar pa te metode praktično nikoli ne srečamo v 3D-igrah, saj je računsko prezahtevna. Se pa uporablja metoda sevanja žarkov, saj je računsko bolj preprosta. Prve 3D-igre so uporabljale prav to metodo za izris na zaslon.

6.1 Implementacija v Digizetu

V Digizetu sem implementiral metodo sevanja žarkov, ki sem jo uporabil za detekcijo vidnih trikotnikov in z njeno pomočjo zgradil PVS ("Potentially visible set") za določeno pozicijo v svetu. Algoritem je sestavljen iz treh delov:

- trk žarka z osmiškim drevesom
- trk žarka s trikotnikom
- sevanje žarkov skozi točke na projekcijski ravnini

V nadaljevanju si bomo pogledali vsakega od teh delov ter zaključili s pregledom rezultatov meritev.

6.1.1 Trk žarka z osmiškim vozliščem

Detekciji trka med žarkom in osmiškim drevesom pravimo tudi obhod drevesa ("Octree Traversing" ali "Octree Traversal"). Cilj je najti vsa tista (končna) vozlišča, ki jih žarek na svoji poti seka. Včasih je pomemben tudi vrstni red vozlišč – ponavadi želimo pravilen vrstni red, kar pomeni, da si vozlišča sledijo eno za drugo v smeri potovanja žarka. Metode za obhod drevesa delimo v dve skupini:

- Od spodaj navzgor ("Bottom-Up Methods"): Obhod začnemo v prvem končnem vozlišču, ki ga žarek seka, nato pa zaporedno iščemo sosednja končna vozlišča, ki jih seka. Vrstni red vozlišč je pravilen (v smislu sledenja žarku).
- Od zgoraj navzdol ("Top-Down Methods"): Obhod začnemo v korenskem vozlišču, nato pa se rekurzivno spuščamo do končnih vozlišč, ki jih seka žarek. V vsakem koraku ugotovimo, katere od osmih otrok trenutnega vozlišča seka žarek, in rekurzijo ponovimo za sekajoča vozlišča. Rekurzijo ustavimo, ko pridemo do končnih vozlišč. Metoda, ki sem jo implementiral v Digizetu, spada v to skupino.

Za konkretno implementacijo v Digizetu sem si izbral algoritem, ki ga je predlagal J. Revelles (in drugi) v [10]. Ta spada v drugo skupino ("Top-Down") metod za obhod drevesa. Tu bom predstavil osnovno idejo postopka, ki je podrobneje opisan v samem članku.

Delovanje algoritma si bomo pogledali na primeru štiriškega drevesa. Posplošitev na osmiško drevo in 3D-prostor je trivialna. Definirajmo žarek r kot par (p, d), kjer je $p = (p_x, p_y)$ izhodišče žarka, $d = (d_x, d_y)$ pa normaliziran smerni vektor žarka. Za vsak $t \geq 0$ lahko na žarku določimo točko $(x_r(t), y_r(t))$, ki jo definiramo kot:

$$\begin{aligned} x_r(t) &= p_x + td_x \\ y_r(t) &= p_y + td_y \end{aligned}$$

Vozlišče o štiriškega drevesa predstavlja množico točk (x, y), kjer je vsaka točka omejena z vrednostmi $x_0(o)$, $x_1(o)$, $y_0(o)$ in $y_1(o)$, ki predstavljajo meje prostora, ki ga definira vozlišče o v x in y smeri. Žarek seka vozlišče o, če obstaja nek $t \geq 0$, pri katerem veljata naslednja pogoja:

$$\begin{aligned} x_0(o) &\leq x_r(t) < x_1(o) \\ y_0(o) &\leq x_r(t) < y_1(o) \end{aligned}$$

Definirajmo vrednosti $t_{x0}(o, r)$, $t_{x1}(o, r)$, $t_{y0}(o, r)$ in $t_{y1}(o, r)$ kot vrednosti t, pri katerih žarek seka meje vozlišča o. Izračunamo jih kot:

$$\begin{aligned} t_{x0}(o, r) &= (x_0(o) - p_x)/d_x \\ t_{x1}(o, r) &= (x_1(o) - p_x)/d_x \\ t_{y0}(o, r) &= (y_0(o) - p_y)/d_y \\ t_{y1}(o, r) &= (y_1(o) - p_y)/d_y \end{aligned}$$

Ko izračunamo vrednosti iz zgornje enačbe, nam preostane le še en korak: preveriti moramo, ali velja naslednja neenakost:

$$max(t_{x0}(o, r), t_{y0}(o, r)) < min(t_{x1}(o, r), t_{y1}(o, r))$$

Če neenakost velja, žarek seka vozlišče. S tem je test zaključen, ne pa še sam algoritem. Za razlago posameznih izpeljav in primer implementacije ter posplošitev na tri dimenzije, naj si bralec ogleda [10]. Algoritem ta test uporablja za testiranje posameznih vozlišč, nato pa skuša rekurzivno testirati tudi otroke pozitivnih vozlišč. V samem članku je razložen točen potek algoritma in različne optimizacije, ki sem jih tudi implementiral.

Prednost tega algoritma je, da pri rekurziji uporablja nekatere vrednosti, ki jih je že izračunal pri starših vozlišča in je zato nekoliko hitrejši od podobnih "Top-Down" algoritmov. Druga prednost pa je njegova preprostost, saj je preprost za implementacijo, nekoliko težji pa za razumevanje. Bilo pa bi zanimivo implementirati še kak drug algoritem iz prve skupine metod za obhod drevesa in primerjati rezultate.

6.1.2 Trk žarka s trikotnikom

Trk žarka s trikotnikom je pomemben test in mora biti izveden hitro, saj ta test izvajamo veliko pogosteje kot test žarka z osmiškim vozliščem. Test mora biti zasnovan tudi tako, da hitro zazna, če do trka ni prišlo ("early reject"),

saj se test izkaže za negativnega pri večini trikotnikov. Za konkretno implementacijo sem si izbral test, ki ga je opisal Möller v [8], ki je eden od najbolj citiranih člankov na to temo, njegov algoritem pa verjetno najbolj znan test med trikotnikom in žarkom. Ta test se od večine drugih razlikuje v tem, da ne potrebuje vnaprej izračunane normale trikotnika in je tudi ne računa sproti. V nadaljevanju bom ta test na kratko opisal.

Definirajmo žarek $R(t)$ z izhodiščem v O in normaliziranim vektorjem smeri D kot:

$$R(t) = O + tD$$

in trikotnik z oglišči V_0, V_1 in V_2. Cilj testa je seveda ta, da ugotovimo, ali dani žarek prebada trikotnik v smeri vektorja smeri. Pri tem želimo tudi preveriti, na kateri strani prebada trikotnik. Če namreč trikotnik prebada na zadnji strani, mora test vrniti negativen rezultat v primeru, ko uporabljamo izločanje zadnjih ploskev.
Opisani test kot rezultat vrne razdaljo t od izhodišča žarka do točke, kjer žarek prebada trikotnik, ter koordinati u in v na trikotniku, kjer je do preboda prišlo.

Točko $T(u, v)$ na trikotniku, kjer sta u in v baricentrični koordinati, izračunamo po enačbi:

$$T(u, v) = (1 - u - v)V_0 + uV_1 + vV_2$$

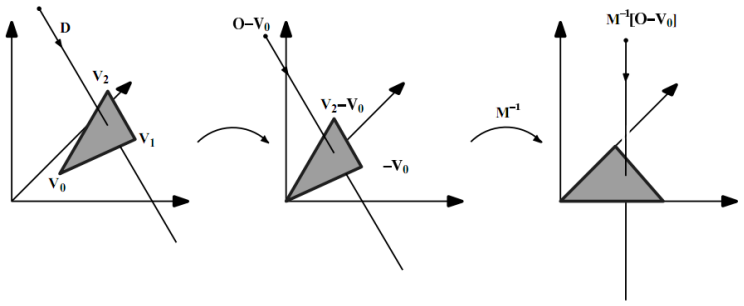
Da bi izračunali sečišče žarka s trikotnikom, moramo izenačiti točko na trikotniku in točko na žarku:

$$O + tD = (1 - u - v)V_0 + uV_1 + vV_2$$

Če ima enačba rešitev, žarek seka trikotnik. Enačbo lahko zapišemo v matrični obliki, primeri za ureditev v sistem linearnih enačb:

$$[-D, V_1 - V_0, V_2 - V_0] \begin{bmatrix} t \\ u \\ v \end{bmatrix} = O - V_0$$

To obliko lahko še nekoliko preuredimo v obliko, primerno za implementacijo. Za podrobnosti in konkretno implementacijo v C-ju glej [8]. Geometričen pomen zadnje enačbe je v tem, da trikotnik transliramo v izhodišče ter v smeri y in z skaliramo dolžine stranic na 1 ter poravnamo žarek z osjo x. To prikazuje slika 6.1. V zadnjem koraku moramo še preveriti vrednost t, saj mora ta biti pozitivna, sicer žarek seka trikotnik v nasprotni smeri vektorja D, česar seveda ne želimo.



Slika 6.1: Trikotnik transliramo v izhodišče in žarek poravnamo z osjo x. Slika je iz [8].

Zastavlja se vprašanje, ali je ta algoritem dejansko najbolj primeren za dani problem? Odgovor je: verjetno ne. Möllerjev test je namreč zasnovan tako, da ne potrebuje normale trikotnika in s tem računa, da bo pridobil na hitrosti pri tem, ko nam ne bo treba shranjevati normale. Manj pomnilnika zasede geometrija, hitreje deluje tudi algoritem, saj pride do manj zgrešitev pri branju iz medpomnilnika. V Digizetu pa hranimo 3 normale na trikotnik, zato bi verjetno bil bolj primeren kak algoritem, ki uporablja vnaprej izračunane normale. Druga stvar, ki jo pri tem testu opazimo, je njegov rezultat. V našem primeru bi zadostoval le rezultat SEKA / NE SEKA, algoritem pa vrne dovolj podatkov, da lahko izračunamo natančno točko, v kateri žarek seka trikotnik. Teh podatkov ne potrebujemo in so odveč. Algoritem, ki bi bil optimiziran samo za ugotavljanje, ali je prišlo do sekanja, bi bil lahko verjetno še malce hitrejši, saj bi se izognili računanju odvečnih podatkov. Tako vidimo, da nam ostaja še nekaj prostora za izboljšave.

6.1.3 Sevanje žarkov skozi točke na projekcijski ravnini

Zadnji korak, ki si ga bomo pogledali, pa je dejansko sevanje žarkov. Pri izrisovanju scene bi morali na projekcijsko ravnino iz očesa (kamere) poslati toliko žarkov, kolikor je pik na zaslonu. Hitrost metode je torej neposredno odvisna od resolucije zaslona oz. slike, na katero izrisujemo sceno. Podobno velja tudi pri izločanju zakritih ploskev, saj je postopek enak, s to razliko, da nam žarka ni treba poslati prav v vsako piko na zaslonu. V Digizetu sem namreč uporabil metodo sevanja žarkov na način, da sem preveril le trke z osmiškimi vozlišči in ne s posameznimi trikotniki. To pomeni, da sem za vsako vozlišče preveril vse trikotnike, vendar pa sem testiranje ustavil ob prvem trikotniku, kjer je test vrnil pozitiven rezultat. Tako vozlišče sem označil kot

vidno. S tem sem se izognil testiranju celotne geometrije in sem test reduciral na vidnost osmiških vozlišč. Druga pomembna poenostavitev pa je, kot že rečeno ta, da žarkov nisem pošiljal prav v vsako točko na zaslonu, ampak sem ustvaril mrežo in žarke pošiljal le na sečišča v mreži. Algoritem sem testiral pri različnih razmakih med točkami. Čim večji je bil razmak, manjša je bila natančnost postopka in več grafičnih artefaktov sem s tem ustvaril (luknje v geometriji ipd). Vendar pa: če pustimo majhen razmak med točkami, so razlike med "popolnim" algoritmom in približnim skoraj neopazne, razlika v pohitritvi pa zelo velika. Za primer vzemimo razmak dveh točk: namesto da bi se med dvema žarkoma premaknili za eno točko v smeri x oz. y, se premaknemo za dve točki. S tem zmanjšamo število žarkov za 4 (v splošnem za n^2, kjer je n razmak med žarki v x oz. y smeri, izražen v številu točk). S tem lahko pričakujemo tudi pohitritev za približno faktor 4. Dejstvi, da taka aproksimacija deluje, je v tem, da testiramo le vidnost osmiških vozlišč, ne pa tudi trikotnikov. Vsako vozlišče namreč lahko vsebuje po več 100 trikotnikov (to je eden od argumentov pri gradnji osmiškega drevesa) in če s testom zadenemo enega od njih, bodo kot vidni označeni vsi, saj je kot vidno označeno vozlišče, ki jih vsebuje. Zato obstaja velika verjetnost, da žarki, ki niso dosti oddaljeni od trenutnega žarka, zadenejo enega od teh trikotnikov istega vozlišča. Z drugimi besedami: verjetnost, da je neko vozlišče označimo kot nevidno, je majhna. Čeprav s testom zgrešimo posamezen trikotnik v vozlišču, lahko še zmeraj zadenemo kakšnega drugega.

Omeniti pa velja še eno optimizacijo pri sevanju žarkov. Če trikotnik, ki ga je žarek zadel, prestavimo na vrh seznama trikotnikov, ki jih vozlišče vsebuje, povečamo verjetnost, da bo pri računanju naslednje sličice test najprej zadel ob ta trikotnik in se bo s tem ustavil prej kot pa v povprečju ("Early accept"). Med sličicami namreč velja lokalnost – pogosto se vidna geometrija med dvema zaporednima sličicama bistveno ne spremeni. Ta lokalnost pa ne velja samo med sličicami, ampak tudi med posameznimi žarki – dva sosednja žarka z veliko verjetnostjo zadeneta ob isti trikotnik zaradi prostorske lokalnosti. In če je ta trikotnik na vrhu seznanama, se bosta oba testa zaključila že po prvem testu trka s trikotnikom. V povprečju sem s to optimizacijo dosegel 11-odstotno pohitritev, kot kaže tabela 6.1.

6.1.4 Rezultati

Metoda sevanja žarkov v splošnem velja za počasno, saj jo praviloma izvajamo na CPE in je zato ne moremo pohitriti z GPE. Drug problem je ta, da je metoda bolj primerna za izrisovanje scene kot izločanje, saj z njeno pomočjo dobimo

Optimizacija	Čas	Delež
ne	74 ms	100%
da	66 ms	89%

Tabela 6.1: Rezultati testa po metodi sevanja žarkov pri dani geometriji, kjer smo enkrat uporabili optimizacijo s premikanjem trikotnikov na vrh seznama, drugič pa ne. V povprečju sem dosegel 11-odstotno pohitritev.

informacijo o barvah vseh slikovnih točk na zaslonu. Te informacije pa pri izločanju ne potrebujemo in je odveč. Lahko bi rekli, da za preprosto opravilo, kot je zabijanje žebljička v steno, uporabljamo macolo namesto kladiva. Vse to, kar nam metoda vrne, ponovno izračunamo, ko izrišemo geometrijo na običajen način na grafični kartici. S tem podvajamo delo in upočasnjujemo pogon. Tudi v mojem primeru se je metoda izkazala za neučinkovito, predvsem v primerjavi z izločanjem s pomočjo poizvedb zakritosti. Kot pa smo videli že v prejšnjih razdelkih, smo pustili precej prostora raznim optimizacijam, ki jih v Digizetu nisem implementiral. V tem poglavju bom predstavil nekaj rezultatov meritev, ki sem jih opravil s to metodo.

Prvi test sem izvajal pri različnih geometrijah in različnih postavitvah kamere. Kot pričakovano se je izkazalo, da je postavitev kamere ključna za učinkovitost metode. Če kamera stoji zelo blizu velikih površin, ki zakrije velik del geometrije, bo izločanje učinkovito. V odprtih prostorih, kjer morajo žarki prepotovati velik del prostora, pa je metoda neučinkovita. Izkaže se, da z metodo v povprečju zmanjšamo število izrisanih sličic na sekundo, torej uporaba izločanja upočasni pogon, kar je ravno obratno od tega, kar želimo. Metoda je uporabna le v zelo redkih primerih, kjer je kamera postavljena pred velike ploskve. Tak primer je prikazan na sliki 6.2, kjer sem test izvajal pri gostoti sevanja 10x10 in sem dosegel obetavne rezultate, ki so prikazani v tabeli 6.2. Večina žarkov je zadela ob bližnje trikotnike, postavljene nekaj metrov pred kamero, zato smo uspeli izločiti skoraj 400.000 trikotnikov. Izločanje je trajalo približno 17 milisekund.

Drugi test sem izvedel na geometriji, ki je vsebovala dva nebotičnika, ki sta skupaj vsebovala 61272 trikotnikov. Test sem izvedel pri štirih različnih gostotah mreže, in sicer 5x5, 10x10, 15x15 ter 20x20 točk. Gostota 5x5 točk pomeni, da smo žarke sevali na 5 zaslonskih točk narazen (tako v x kot y smeri). Številka pravzaprav pomeni razmak v x in y smeri med posameznimi žarki. V test sem vključil dve različni drevesi, ki sem ju zgradil z različnimi parametri. Rezultate prikazuje tabela 6.3.

Izločanje	FPS	Št. vidnih trikotnikov	Čas izrisovanja sličice
ne	28	382080	35,7 ms
da	48	302	3,6 ms

Tabela 6.2: Rezultati testa na geometriji s slike 6.2. Izločanje je trajalo 17,2 ms, vendar je znižalo čas za izris slike s 35,7 na 3,6 ms.

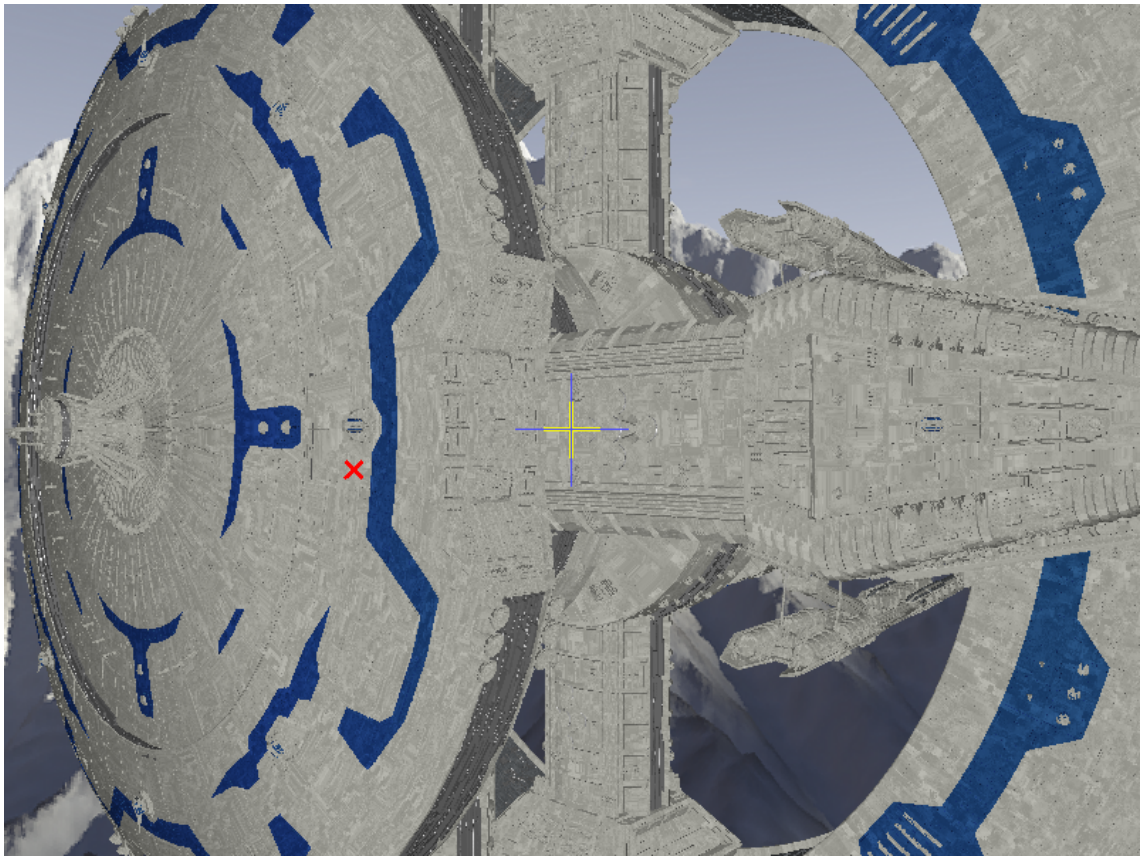
Drevo	5x5	10x10	15x15	20x20
1	196 ms	50 ms	23 ms	14 ms
2	2681 ms	663 ms	314 ms	172 ms

Tabela 6.3: Rezultati meritve časa, potrebnega za detekcijo zakritih ploskev pri dani geometriji in poziciji kamere. Prvo drevo smo zgradili z radijem 2048 ter maksimalnim številom trikotnikov v končnem vozlišču 50, pri drugem drevesu pa smo uporabili radij 11000 ter število trikotnikov 150. Rezultati so merjeni pri štirih različno gostih mrežah sevanja žarkov.

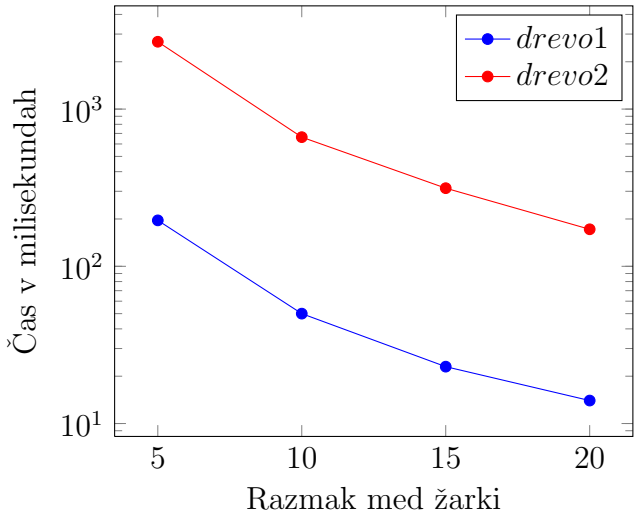
Razmerje med hitrostjo in gostoto mreže bolj nazorno prikazuje graf na sliki 6.3. Vidimo, da je to razmerje približno premo sorazmerno. To smo tudi pričakovali (glej razdelek 6.1.3). Druga pomembna stvar, ki jo opazimo, pa je, da je razlika v hitrosti pri isti geometriji in poziciji kamere za različni drevesi zelo velika. Pri drugem drevesu smo uporabili radij, ki je bil veliko večji od prvega. To pomeni, da bodo tudi najmanjša končna vozlišča precej večja kot pri prvem drevesu. Poleg tega smo dovolili, da je v končnih vozliščih več trikotnikov, kar pomeni, da se bodo vozlišča manj delila, tako pa bodo končna vozlišča tudi večja. Iz tega lahko sklepamo, da bo glavnino časa algoritem prebil v testu za zaznavanje trka med žarkom in trikotnikom. Ta test pa, kot smo videli v razdelku 6.1.2 na strani 45, ni poceni. V primeru prvega drevesa pa bolj pogosto izvajamo test med žarkom in osmiškim vozliščem. Tako veliko razliko v hitrosti lahko razložimo tudi takole: če ne bi postavili omejitve za maksimalno število trikotnikov v končnem vozlišču, se drevo sploh ne bi delilo in vsi trikotniki bi bili v korenskem vozlišču. To pa pomeni, da bi drevo izgubilo svoj pomen in bi dobili situacijo, podobni tisti, ko ne uporabljamo drevesa. Zato lahko razumemo, zakaj je pri manj razdeljenem drevesu metoda sevanja žarkov tudi počasnejša – preverjati moramo namreč veliko množico trikotnikov, namesto da bi preverili le trikotnike na poti žarka.

Iz povedanega sledi, da je uporaba metode sevanja žarkov v Digizetu vprašljiva.

Kot bomo videli v poglavju o poizvedbah zakritosti, so te v našem primeru veliko bolj primerne. Razlika v rezultatih med enim in drugim drevesom pri isti geometriji in postavitvi kamere kaže, da je izbira parametrov pri gradnji osmiškega drevesa ključna za učinkovito delovanje metode. Pri določenih tipih geometrije bi se verjetno lahko zgodilo, da bi bila ta metoda zelo učinkovita. Vendar pa to v primeru Digizeta, kjer uporabljamo odprte, prostrane scene, ni tako.



Slika 6.2: Prikaz scene, nad katero sem izvajal test z metodo sevanja žarkov. Geometrija je sestavljena iz ∼1,5 M trikotnikov, premer vesoljske postaje pa je približno 10 kilometrov. Kamera je bila postavljena nekaj metrov nad trupom ladje v točki, označeni z rdečim križcem ter obrnjena proti drugemu koncu postaje (desna stran slike).



Slika 6.3: Grafični prikaz meritev iz tabele 6.3. Za čas je uporabljena logaritmična skala. Odvisnost med gostoto mreže in hitrostjo je približno linearna.

Poglavje 7

Izločanje s pomočjo poizvedb zakritosti

Poizvedbe zakritosti ("Occlusion queries") so orodje, ki ga ponujajo moderne grafične kartice. Ideja je, da bi grafična kartica sama izračunala, ali je nek objekt viden. To naredimo tako, da izrišemo vseobsegajoče telo nekega objekta s poizvedbami zakritosti, pri čemer se objekt dejansko ne izriše na zaslon, ampak ga grafična kartice le skuša izrisati tako, da za vsako točko preveri, ali opravi Z-buffer test. Pri tem prešteje število točk, ki so test opravile in to posreduje nazaj kot rezultat poizvedbe. Če je število teh točk enako 0 oz. je manjše od nekega praga, objekta ne izrišemo. Na ta način prihranimo pošiljanje geometrije na grafično kartico, ki jo ta objekt obsega. V Digizetu za vseobsegajoča telesa uporabljam osmiška vozlišča. Osmiško vozlišče ima obliko kocke, zato ni drago za izris. Če ugotovimo, da neko vozlišče ni vidno, lahko izločimo vse trikotnike, ki jih vsebuje.

Skozi čas so se strojne implementacije poizvedb spreminjale. V OpenGL lahko na primer uporabimo:

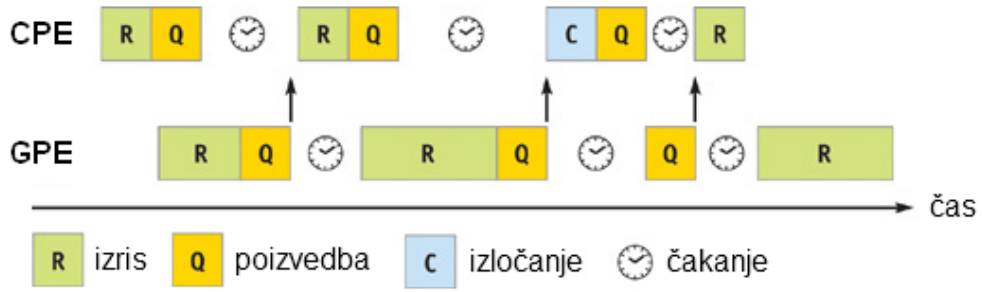
- HP_occlusion_test
- NV_occlusion_query
- ARB_occlusion_query

Prva različica, HP_occlusion_test, ni omogočala izvajanja več poizvedb naenkrat. Prav tako je kot rezultat vrnila le TRUE/FALSE vrednost in ne števila vidnih pik. NV in ARB različici pa to že omogočata. Najnovejšo različico, ARB_occlusion_query, sem tudi uporabil pri implementaciji v Digizetu.

Poizvedbo zakritosti v splošnem uporabljamo na naslednji način:

1. Ustvarimo poizvedbo.
2. Onemogočimo izrisovanje na zaslon.
3. Onemogočimo pisanje v globinski medpomnilnik ("depth buffer").
4. Vklopimo poizvedbo (s tem resetiramo števec vidnih pik).
5. Izrišemo vseobsegajoče telo brez tekstur in senčenja (ta korak želimo čimbolj pohitriti). Pri tem se telo dejansko ne izriše, saj smo onemogočili risanje na zaslon in v globinski medpomnilnik.
6. Zaključimo poizvedbo (s tem ustavimo štetje vidnih pik).
7. Omogočimo izrisovanje na zaslon.
8. Omogočimo pisanje v globinski medpomnilnik.
9. Preberemo rezultat poizvedbe.
10. Če je število vidnih pik večje od 0 oz. neke meje, izrišemo vso geometrijo, ki jo vsebuje vseobsegajoče telo.

Pri taki uporabi pa se pojavi problem: GPE potrebuje nekaj časa, da izriše dano telo ter prešteje vidne pike in sporoči rezultate uporabniku. Med tem časom je GPE nedostopna. To pa pomeni, da tudi na strani CPE ne delamo ničesar, saj čakamo na rezultate z GPE. To je znan problem pri poizvedbah zakritosti in se ga rešuje na različne načine. Če ga ne rešujemo, se lahko zgodi, da izločanje s poizvedbami pogon celo upočasni. Problem rešujemo na splošno tako, da med časom, ko GPE izvaja poizvedbo, na GPE pošiljamo vidno geometrijo ali računamo druge stvari. Pri novejši različicah poizvedb lahko GPE tudi povprašamo, ali je določena poizvedba že končana. Pri bolj sofisticiranih algoritmih uporabljamo tudi to informacijo, saj v splošnem težko predvidimo, koliko časa bo vzela poizvedba. Druga pomembna stvar pa je, da običajno izvajamo več poizvedb naenkrat. Tudi to omogočajo novejše različice poizvedb zakritosti (recimo ARB). To je pomembno predvsem zato, da lahko hkrati preverimo vidljivost več vozlišč v drevesu in ne le enega. S tem lahko skrajšamo skupni čas čakanja na GPE, saj naenkrat preberemo rezultate vseh poizvedb. Če bi izvajali vsako poizvedbo posebej, bi morali vmes kar naprej čakati na podatke od GPE in bi se zakasnitve zaradi prenosa podatkov med posameznimi poizvedbami seštevale. Tako pa imamo le eno skupno zakasnitev za vse opravljene poizvedbe. Primer naivne uporabe poizvedb zakritosti prikazuje slika 7.1.



Slika 7.1: Prikaz situacije pri naivni uporabi poizvedb zakritosti. Tako CPE kot GPE velik del časa ne počneta ničesar. Slika povzeta po [5].

7.1 Implementacija v Digizetu

V Digizetu sem poizvedbe zakritosti primarno uporabil za računanje PVS, ki sem ga nato uporabil za izločanje nevidnih trikotnikov. Poleg tega sem implementiral še naivno različico poizvedb zakritosti, ki vidljivost računa sproti, za vsako sličico posebej. Te implementacije nisem posebno optimiziral. V nadaljevanju bom predstavil obe različici ter rezultate meritev.

7.1.1 Implementacija naivne, realnočasovne metode

Realnočasovna metoda, ki sem jo implementiral, je naivna iz dveh razlogov:

- ker na zelo poenostavljen način izbira potencialno dobre zakrivalne ploskve;
- ker ne odpravi problema s čakanjem rezultatov poizvedb z GPE.

Algoritem je sestavljen iz naslednjih korakov:

1. Poiščemo vsa končna osmiška vozlišča, ki so vidna (so znotraj piramide pogleda) in jih razvrstimo po oddaljenosti. Pri tem oddaljenost izračunamo tako, da vzamemo središče osmiškega vozlišča ter izračunamo običajno geometrično razdaljo do kamere.
2. Vzamemo 20% najbližjih vozlišč in jih izrišemo. Ta vozlišča predstavljajo potencialno dobre zakrivalne objekte.
3. Ustvarimo poizvedbe za preostalih 80% vozlišč in počakamo na rezultate.

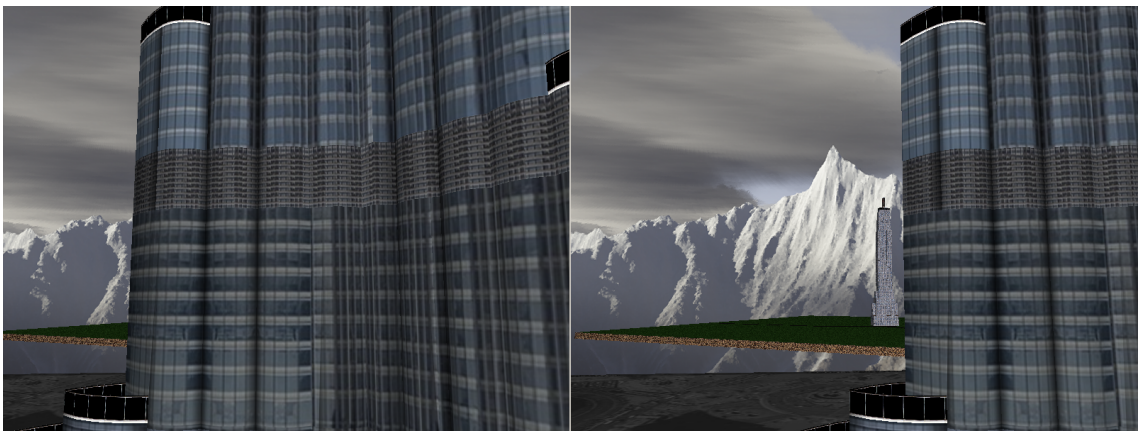
4. Od preostalih 80% vozlišč izrišemo tista, za katere smo s poizvedbo ugotovili, da so vidna.

Kot vidimo iz delovanja algoritma, je izbira potencialno dobrih zakrivalnih ploskev zelo preprosta. Algoritem bi lahko izboljšali na primer tako, da bi že pri gradnji osmiškega drevesa izračunali, katera vozlišča lahko dobro zakrivajo ostale objekte. Pri tem bi za mero vzeli recimo skupno površino trikotnikov, ki jih vsebuje vozlišče. Druga taka mera pa bi bila oddaljenost vozlišča od kamere. Vozlišča, ki so bližja kameri, lahko zakrijejo večji del prostora kot tista, ki so dlje.

Metodo sem preiskusil nad sceno z dvema nebotičnikoma, ki skupaj vsebujeta 61272 trikotnikov. Slika 7.2 prikazuje dve različni poziciji kamere. Iz prve pozicije je zakrita večina geometrije, iz druge pa je večina geometrije vidna. Rezultate meritve prikazuje tabela 7.1.

Scena	Izločanje	FPS	Št. vidnih trikotnikov
1	ne	168	57202
1	da	333	6750
2	ne	168	57202
2	da	137	55024

Tabela 7.1: Rezultati meritve na geometriji s slike 7.2.



Slika 7.2: Različni poziciji kamere pri meritvi učinkovitosti naivne metode izločanja. Rezultati meritve so podani v tabeli 7.1. Na levi sliki je nebotičnik, ki je sestavljen iz približno 50.000 trikotnikov, zakrit, na desni pa viden.

Kot vidimo iz rezultatov meritve, je učinkovitost metode zelo odvisna od pozicije kamere. Če so pred nami ploskve, ki zakrijejo večji del preostale geometrije, se izločanje izplača. Če pa je zakrite le malo geometrije, bo izločanje upočasnilo pogon. V povprečju pa je izločanje s pomočjo poizvedb zakritosti v najslabših primerih manj upočasnilo pogon, kot pri uporabi metode sevanja žarkov. Še ena možna optimizacija bi bila na primer ta, da bi dve zaporedni sličici izračunali enkrat z izločanjem, drugič pa brez. Pri tem bi merili čas, potreben za izris. Če bi bil čas pri drugi sličici krajši, bi izločanje za nekaj časa izklopili, v nasprotnem primeru pa bi nekaj časa vse sličice izrisovali z vklopljenim izločanjem.

7.1.2 Implementacija celične metode

Pri celičnem pristopu skušamo vidljivost izračunati za določeno celico (podprostor). Celico lahko predstavlja vozlišče v drevesu, ali pa jo izberemo na kak drug način. V Digizetu sem za celico izbral sfero s polmerom 0,5 m.

Najbolj znan primer celičnih metod je izločanje s portali, ki ga uporablja veliko modernih iger in smo ga spoznali v poglavju 3. Ideja celičnih metod je, da izračunamo vidljivost vnaprej, med izvajanjem programa pa le preberemo informacijo o vidljivosti iz vnaprej pripravljene tabele. Na ta način se izognemo velikokrat dragemu preračunavanju vidljivost v realnem času. Pri branju te informacije iz tabele pa kazni za računanje vidljivosti v slabo zakriti sceni praktično ni.

Vidljivost je iz katerekoli točke v celici enaka. To pomeni, da je PVS, zgrajen za celotno celico, bolj konzervativen kot za posamezne točke znotraj celice. Če bi vidljivost računali v vsaki točki posebej, bi v splošnem lahko izločili še več geometrije. Vendar pa bi za to porabili mnogo več pomnilnika in v praksi to ne bi bilo izvedljivo.

Prvi test, ki sem ga implementiral, je test vidljivosti celotnega vidnega prostora okoli kamere. Naloga je takšna: za dano točko (x, y, z) izračunaj PVS za poljubno orientacijo kamere. To lahko preprosto realiziramo tako, da postopoma rotiramo kamero okoli XZ in YZ ravnin ter računamo vidljivost za vsako pozicijo posebej. Na koncu naredimo unijo PVS-jev posameznih sličic, kar nam da željeni PVS v dani točki. Psevdo koda implementacije je takšna:

```
IzklopiIzrisovanje;
PVS.Clear;
FOR (int v = -180/fovy/2; v < 180/fovy/2; v++)
BEGIN
  FOR (int h = -360/fovx/2; h < 360/fovx/2; h++)
```

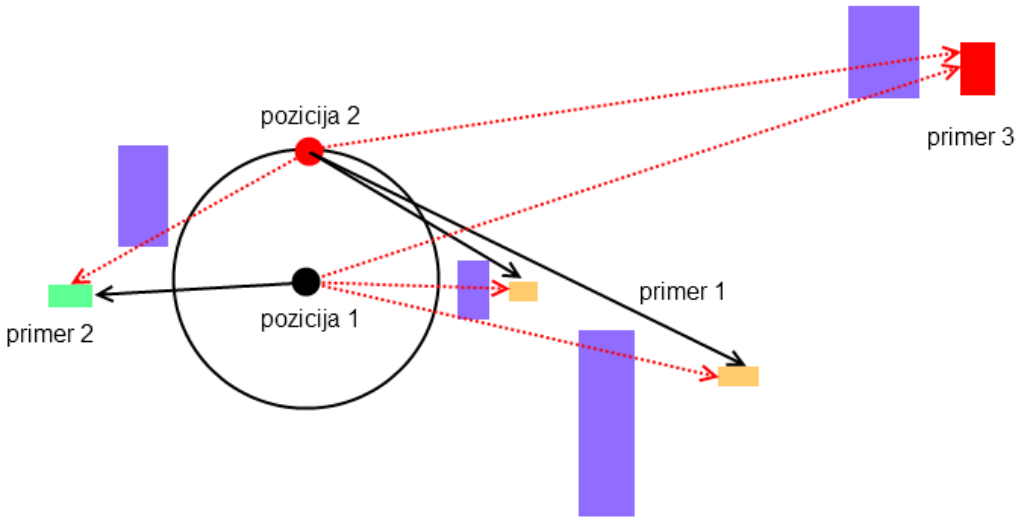
```
  BEGIN
    Camera.angleXZ = h * fovx;
    Camera.angleYZ = v * fovy;
TempPVS = IzvediTestVidljivosti(Camera);
ResetirajZBuffer;
PVS = PVS UNION TempPVS;
  END
END
```

Tako izračunamo PVS za poljubno orientacijo kamere v dani točki. Vidljivost za posamezno pozicijo izračunamo na način, podoben tistemu, ki smo ga opisali v razdelku 7.1.1. Razlika je v tem, da tu iščemo eksaktni PVS, torej kar EVS. To storimo s pomočjo poizvedb zakritosti. Ta korak bi lahko opravili tudi na kak drug način, recimo s pomočjo metode sevanja žarkov, vendar se izkaže metoda s poizvedbami veliko hitrejša in natančnejša. Če bi hoteli z metodo sevanja žarkov doseči enako natančnost, bi morali žarke poslati v prav vsako piko na projekcijski ravnini. Kot pa smo to spoznali v poglavju 6, tega pri hierarhičnem izločanju običajno ne delamo. Postopek računanja vidljivosti je takšen:

1. Izberemo vsa vidna, končna osmiška vozlišča in jih uredimo po oddaljenosti.
2. Za vsako vozlišče ustvarimo poizvedbo in jo aktiviramo.
3. Izrišemo vsa vozlišča v pravilnem vrstnem redu (glede na oddaljenost od kamere).
4. Preberemo rezultate poizvedb in vozlišča, ki niso nič prispevala k sliki, označimo kot nevidna. Ostala vozlišča dodamo v PVS.

Na ta način izračunamo PVS, ki se natančno ujema z EVS. Sedaj pa sledi še zadnji korak postopka, to je vnaprejšnja priprava celičnih PVS-jev za uporabo med izvajanjem programa. Ta korak se je izkazal za najbolj zahtevnega. Med testiranjem sem ugotovil, da bi za konstruiranje celic program porabil preveč časa, da bi jih izračunal v nekem doglednem času. Zato sem ta korak poenostavil do te mere, da sem vsako novo točko, v kateri sem izračunal PVS, razglasil za celico v obliki sfere s polmerom 0,5 m. Tako sem se izognil dragemu preračunavanju PVS-ja v okolici te točke. Slabost tega pristopa je, da dlje, ko smo stran od središča celice (torej od točke, v kateri je PVS izračunan), večja je verjetnost, da bo del izrisane geometrije v resnici zakrit ali pa bomo izpustili izris dela geometrije, ki je viden, vendar smo ga v središču celice razglasili za nevidnega. Večji polmer sfere, ko uporabimo, bolj približen postane PVS in s

tem je večja verjetnost, da bo pri izrisu prišlo do vidnih artefaktov. Vrednost 0,5 m se je v mojem primeru izkazala za sprejemljivo. Če v bližini celice ni veliko ploskev, je nevarnost za ustvarjanje artefaktov manjša. To prikazuje slika 7.3, primer 3. PVS je enak EVS le natanko v središču sfere. Dlje, ko gremo iz središča, manj natančen je PVS. Slika v primeru 1 prikazuje dva objekta, ki sta vidna iz robne točke, iz središčne pa ne. V primeru 2 pa prikazuje obrnjeno situacijo: objekt je viden iz središčne točke, iz robne pa ne. V enem primeru je PVS preveč konzervativen, v drugem pa preveč agresiven (o tipih PVS-jev glej poglavje 3.1).



Slika 7.3: Celica v obliki sfere. V poziciji 1 je PVS enak EVS, v poziciji 2 pa je le še približek. V primerih 1 in 2 vidimo, da se PVS na obrobju celice ne ujema z EVSjem. Tretji primer pa prikazuje oddaljen objekt, ki ne more bistveno vplivati na poslabšanje robnega PVS-ja.

Sedaj pa si poglejmo še problem vnaprejšnjega izračuna celic, za katerega smo rekli, da se izkaže za računsko zelo zahtevnega. Če hočemo poznati vidnost iz katerekoli pozicije v svetu, moramo izračunati PVS-je na primer nad gosto mrežo točk. Vzemimo, da zadostuje razmak 0,5 m med posameznimi točkami. Pri tem razmaku že lahko prihaja do vidnih artefaktov, zato bi za večjo natančnost morali vzeti še bolj gosto mrežo. Vzemimo, da je premer sveta enak 1000 m, torej je prostor velik 1 km^3. Domnevajmo, da je lahko kamera kjerkoli v svetu. Zato moramo izračunati vidljivost v $(2 \cdot 1000)^3 = 8000000000 = 8 \cdot 10^9$

celicah. Recimo, da je vertikalni vidni kot kamere enak 45 °, horizontalni pa 60 ° (to je realni primer iz Digizeta). Izračunati moramo torej PVS v $(180/45) \cdot (360/60) = 4 \cdot 6 = 24$ sličicah. Za lažje računanje domnevajmo, da pogon v povprečju prikazuje 24 sličic na sekundo. Ker je ugotavljanje vidljivosti v eni sličici časovno približno enako potratno kot izris posamezne sličice, porabimo za izračun PVS v eni točki eno sekundo. Za izračun mreže PVS bi potrebovali $8 \cdot 10^9$ sekund, kar je enako $92592,6$ dnevom oz. približno enako 257 letom. Vidimo, da je problem v praksi nerešljiv. Seveda je mogoče takšno računanje pohitriti, vendar pa problem vseeno ostaja prezahteven. Četudi bi uspeli izračunati vidljivost v vseh točkah, bi imeli problem, kako shraniti rezultate v pomnilnik, od koder bi jih brali med izvajanjem programa. Zato se vidljivost zmeraj računa nad celicami, le te pa se določa po različnih algoritmih, ki so večinoma zelo zapleteni in jih tu ne bomo obravnavali. V Digizetu sem problem poenostavil na računanje v točkah, ki so najbolj pogosto obiskane. Te točke dobimo tako, da se nekajkrat sprehodimo po sceni in beležimo opravljeno pot. Nato izračunamo PVS le v točkah, ki ležijo na poti. Če med izvajanjem programa obiščemo točko, ki ni bila vnaprej izračunana, jo izračunamo sproti in jo shranimo v medpomnilnik. Če je točk preveč, lahko stare (najdalj neobiskane) zavržemo in s tem sprostimo prostor za nove točke. Kazen za izračun nove točke ni velika – domnevamo namreč, da kamera ne spreminja lege prav hitro. To pomeni, da nam ne bo treba izračunati PVS-ja v novi točki toliko časa, dokler bomo od točke oddaljeni manj kot 0,5 m. Če bi bila hitrost gibanja kamere enaka 1 m/s, bi morali v najboljšem primeru izračunati novo točko vsako sekundo, v najslabšem pa vsake pol sekunde.

V tabeli 7.2 predstavljam rezultate meritev hitrosti izrisovanja s pomočjo celične metode, pri čemer sem vidljivost izračunal vnaprej. Meritev sem opravil pri isti geometriji in v istih točkah kot pri meritvi iz tabele 7.1. V tabelo sem vključil še rezultate iz meritve z naivno metodo iz poglavja 7.1.1 (rezultati tabele 7.1).

Vidimo, da daje ta metoda najboljše rezultate. To smo tudi pričakovali, saj smo vidljivost izračunali že vnaprej in smo se tako v izvajalnem času izognili dragim kaznim v primeru, ko scena ni dobro zakrita. Branje shranjenih PVS-jev iz pomnilnika nas namreč stane zanemarljivo malo. Slabosti metode pa smo že spoznali. Na splošno je skorajda nemogoče izračunati natančno vidljivost v vseh možnih točkah. Zato se v praksi zmeraj zadovoljimo z rešitvami, ki vrnejo le približne PVS-je.

Scena	Izločanje	FPS	Št. vidnih trikotnikov
1	brez	168	57202
1	realnočasovno	333	6750
1	celično	649	6700
2	brez	168	57202
2	realnočasovno	137	55024
2	celično	184	54198

Tabela 7.2: Rezultati meritve na geometriji s slike 7.2 pri dveh različnih metodah izločanja.

Poglavje 8

Zaključek

V diplomski nalogi smo si ogledali nekatere metode izločanja. Ogledali smo si konkretno implementacijo štirih metod ter analizirali vsako posebej ter predstavili rezultate meritev. Od štirih metod dve spadata v ločeno skupino metod izločanja nevidnih ploskev, dve pa spadata v skupino metod izločanja zakritih ploskev. Poleg tega smo si pogledali pojem grafa scene in tri različna drevesa, s pomočjo katerih implementiramo graf scene. Bolj natančno smo si pogledali osmiško drevo, ki sem ga tudi implementiral v pogonu in na katerem temeljijo tri izmed štirih implementacij metod izločanja. Spoznali smo, da je pravilna izbira parametrov pri gradnji drevesa ključna za učikovitost izločanja.

Izločanje zadnjih ploskev in izločanje predmetov izven piramide pogleda sta dve metodi, ki sta preprosti za implementacijo, izločita pa (običajno) največji del geometrije. Videli smo, da z izločanjem zadnjih ploskev izločimo približno 50% vseh ploskev, z izločanjem izven piramide pogleda pa v povprečju še veliko več. Medtem, ko dandanes uporabljamo strojne implementacije izločanja zadnjih ploskev, pa izločanje objektov izven piramide pogleda vedno implementiramo v samem pogonu, saj je metoda vezana na graf scene (praviloma uporabimo hierarhično izločanje). Vedno težimo k temu, da na grafično kartico pošljemo čim manj primitivov, ker je prenos med CPE in GPE pogosto ozko grlo pogona. Izločanja zadnjih ploskev na programski način bistveno ne moremo pohitriti, čeprav je v teoriji to mogoče. Iz meritev smo videli, da bi lahko pogon še nekoliko pohitrili, če bi za vsak trikotnik na preprost način ugotovili, ali je obrnjen proč od nas in bi ga tako izločili. Vendar pa nas v praksi test, ali je trikotnik obrnjen proč, vedno nekaj stane in težko dosežemo pohitritev na tak način. Omenili smo tudi članek, v katerem je predstavljena še ena možna pohitritev, ki skuša poligone grupirati v grupe in naenkrat ugotoviti orientacijo celotne grupe. Pri izločanju izven piramide pogleda smo

si pogledali dve možni optimizaciji, ki sem jih tudi implementiral in testiral. Izkaže se, da lahko z njima dosežemo še majhno pohitritev. Učinkovitost metode pa je predvsem odvisna od pravilne gradnje in uporabe drevesa, nad katerim postopek izvajamo.

Poudarek v diplomi je bil na tehnikah izločanja zakritih ploskev, od katerih smo si pogledali metodo sevanja žarkov ter uporabo poizvedb zakritosti. Pri sevanju žarkov smo ugotovili, da ta metoda ni najbolj primerna kot tehnika izločanja, se pa uporablja v druge namene. Analizirali smo tudi implementacijo in ugotovili, da je še precej možnosti za izboljšave. Predvsem bi lahko poskusil implementirati še kakšno drugo metodo za test trka žarka s trikotnikom, preizkusil pa bi lahko tudi kak algoritem, ki žarke računa vzporedno in izkorišča njihovo prostorsko lokalnost. Namreč dva sosednja žarka ponavadi zadeneta ista vozlišča in celo isti trikotnik znotraj vozlišča. Pri poizvedbah zakritosti smo si pogledali dve različni uporabi metode in obe različici algoritma analizirali. Pri naivni implementaciji realnočasovne različice smo v določenih primerih dosegli pohitritev, v splošnem pa ne. Videli smo, da je glavni problem 'stradanje GPE ("GPU starvation"). To pomeni, da veliko časa GPE ne dela ničesar temveč čaka na ukaze od CPE. Prav tako CPE veliko časa porabi za čakanje rezultatov poizvedb z GPE. Omenili smo, da problem rešujemo tako, da čas, ki ga CPE porabi za čakanje, zapolnimo z drugi opravili. Ena možnost je ta, da CPE v več korakih izvede poizvedbe, vmes pa pošilja v izris tudi del geometrije, za katerega sumi, da je viden. En primer hevristike, ki ocenjuje, kateri deli sveta so vidni, je sledeč: vsa vozlišča, ki so bila vidna v prejšnji sličici, so zelo verjetno vidna tudi v trenutni. Možnosti za izboljšave tega algoritma je veliko, na primer [1]. Pri celični različici, kjer smo vidnost izračunali vnaprej in velja za določeno celico (podprostor), pa smo dosegli odlične rezultate. V povprečju smo pohitrili izrisovanje za več faktorjev. Pohitritev pa je povsem odvisna od geometrije. Če si predstavljamo steno, za njo pa na milijone trikotnikov, je lahko pohitritev praktično neomejena. Po drugi strani pa si lahko predstavljamo sceno, kjer nobena ploskev ni zakrita. V tem primeru pohitritve ni. Ugotovili smo tudi, da računanje vidljivosti za vsako točko v svetu ni praktično izvedljiv, saj vzame preveč časa. Zato se moramo poslužiti drugih postopkov, kako zgraditi celice, za katere izračunamo vidljivost.

Slike

Tabele

Literatura

[1] J. Bittner, M. Wimmer, H. Piringer, W. Purgathofer, "Coherent Hierarchical Culling: Hardware Occlusion Queries Made Useful," v zborniku *Proceedings of Eurographics 2004*, 23(3), str. 615-624. Dostopno na: http://www.cg.tuwien.ac.at/research/vr/chcull/index.html

[2] (1998) BSP Tree Frequently Asked Questions (SGI). Dostopno na: ftp://ftp.sgi.com/other/bspfaq/faq/bspfaq.html

[3] D. Eberly, *3D game engine design: a practical approach to real-time computer graphics*, Morgan Kaufmann Publishers Inc., San Francisco, CA, 2000.

[4] S. Ranta-Eskola, "Binary Space Partitioning Trees and Polygon Removal in Real Time 3D Rendering", magistrska naloga, Uppsala, Švedska, 2001. Dostopno na: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.110.3498&rep=rep1&type=pdf

[5] M. Pharr, R. Fernando, "GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation", Addison-Wesley, 2005, pogl. 6. Dostopno na: http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter06.html

[6] A. Johanssen, M. Carter, "Clustered backface culling," v zborniku *Journal of Graphics Tools*, 3(1):1-14, 1998. Dostopno na: http://zach.in.tu-clausthal.de/teaching/cg_literatur/clustered_backface_culling.pdf

[7] J. Malik, "Human Visual System", zapiski s predavanj 27. Januar 2004. Dostopno na: http://www.cs.berkeley.edu/∼malik/cs294/lecture2-RW.pdf

[8] T. Möller, B. Trumbore, "Fast, Minimum Storage Ray/Triangle Intersection," v zborniku *Journal of Graphics Tools*, 2(1):21–28, 1997. Dostopno na: http://www.graphics.cornell.edu/pubs/1997/MT97.html

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

[10] J. Revelles, C. Urena, M. Lastra, "An efficient parametric algorithm for octree traversal," v zborniku *WSCG 2000 Conference*, Pilsen, Czech Republic, feb. 2000, str. 212–219. Dostopno na: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.29.987&rep=rep1&type=pdf

[11] D. Sýkora, J. Jelínek, "Efficient View Frustum Culling," v zborniku *Proceedings of the 6th Central European Seminar on Computer Graphics (CESCG'02)*, str. 55-64, Budmerice, Slovaška, april 2002. Dostopno na: http://gv2.cs.tcd.ie/sykora/

[12] I. Wald, V. Havran, "On building fast kd-trees for ray tracing, and doing that in O(N log N)," v zborniku *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, 2006, str. 61-69. Dostopno na: http://www.sci.utah.edu/∼wald/Publications/

[13] (2010) Wikipedia - kd tree. Dostopno na: http://en.wikipedia.org/wiki/Kd-tree