SVEUČILIŠTE U ZAGREB

FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD 1855

ALGORITMI ZA UKLANJANJE POLIGONA

Vedran Vukelić

Zagreb, srpanj 2010.

Sažetak

U ovom radu se prvo osvrćem na temu grafa scene. Kratko opisujem ideju koja stoji iza takve strukture koristeći iskustvo koje sam stekao radeći vlastitu aplikaciju po uzoru na OpenSceneGraph. U poglavlju tri sam predstavio nekoliko načina dijeljenja prostora scene , a u četvrtom poglavlju stoji podjela i pregled algoritama za odbacivanje suvišnih poligona. U poglavlju pet sam napravio analizu optimizacija iscrtavanja.

1. Uvod

Graf scene je podatkovna struktura koja ima veliku primjenu u današnjim programima za grafiku i 3D modeliranje. Također je zastupljen u mnogim računalnim igrama. AutoCAD, Adobe Illustrator, Acrobat 3D, CorelDRAW, OpenSceneGraph i OpenSG su samo neki od programa koji koriste graf scene i tu se jasno vidi njegova opće prihvaćena upotreba i svojevrstan standard kada pričamo o strukturi virtualnih scena.

Vječita borba vodi se između sklopovskih mogućnosti i kompleksnosti virtualnih prostora. Svako poboljšanje sklopovske opreme prati želja za što detaljnijom scenom s objektima od pregršt poligona radi veće uvjerljivosti virtualnog prostora. Iz tog razloga postoje algoritmi za optimizaciju iscrtavnja. To su algoritmi koji ”traže” vidljive poligone odnosno određuju nevidljive poligone. Njihov cilj je ubrzanje iscrtavanja, dok scena ostaje jednako impresivna. Jednadžba je jednostavna - što manje poligona pošaljemo u grafički protočni sustav, veća je brzina iscrtavanja (engl. FPS – frames per second).

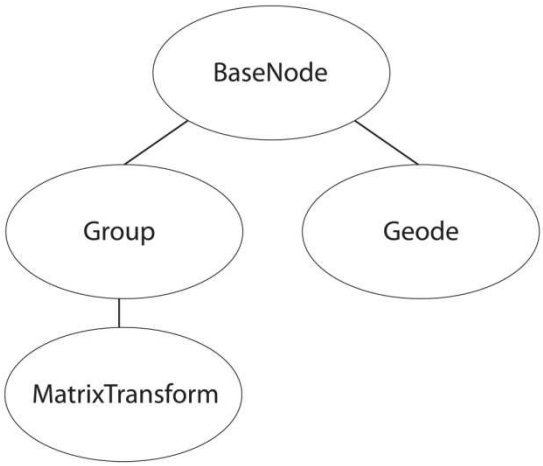
2. Graf scene

Graf scene je hijerarhijska struktura organizirana kao stablo, osmišljena radi što boljeg spremanja scene, njenog iscrtavanja i logičke organizacije. Sadrži u sebi podatke o prostornim transformacijama, izgledu predmeta, obujmicama te vrhovima tj. samoj geometriji predmeta. [1]

2.1. Struktura grafa scene i prolasci kroz graf

Graf scene se sastoji od povezanih čvorova. Svaki graf scene počinje sa korjenom, nakon kojeg dolaze tzv. grupni čvorovi. To su čvorovi koji mogu imati djecu, dakle sama srž grafa scene, ali i mogu imati informacije o prostornim transformacijama. I na kraju imamo listove tj. završne čvorove koji sadrže geometriju tijela. [2]

Na slici 1 je prikazano kako je to organizirano u OpenSceneGraph formatu koji sam i ja koristio kao referencu za izradu vlastitog grafa scene.



Slika 1. Organizacija grafa scene u programskoj implementaciji.

BaseNode je bazni virtualni razred naslijeđen od svih čvorova. Sadrži informacije i funkcije koje su zajedničke svim čvorovima. To su npr. informacije o roditelju, obujmice…

Group je osnovni razred za sve čvorove koji mogu imati djecu. Sadrži sve funkcije za dodavanje i brisanje i naravno samu listu djece.

MatrixTransform naslijeđuje Group, a sadrži informacije o prostornim transformacijama, dakle o rotaciji, translaciji i skaliranju.

Geode nasljeđuje BaseNode i sadrži geometriju tijela u lokalnim koordinatama.

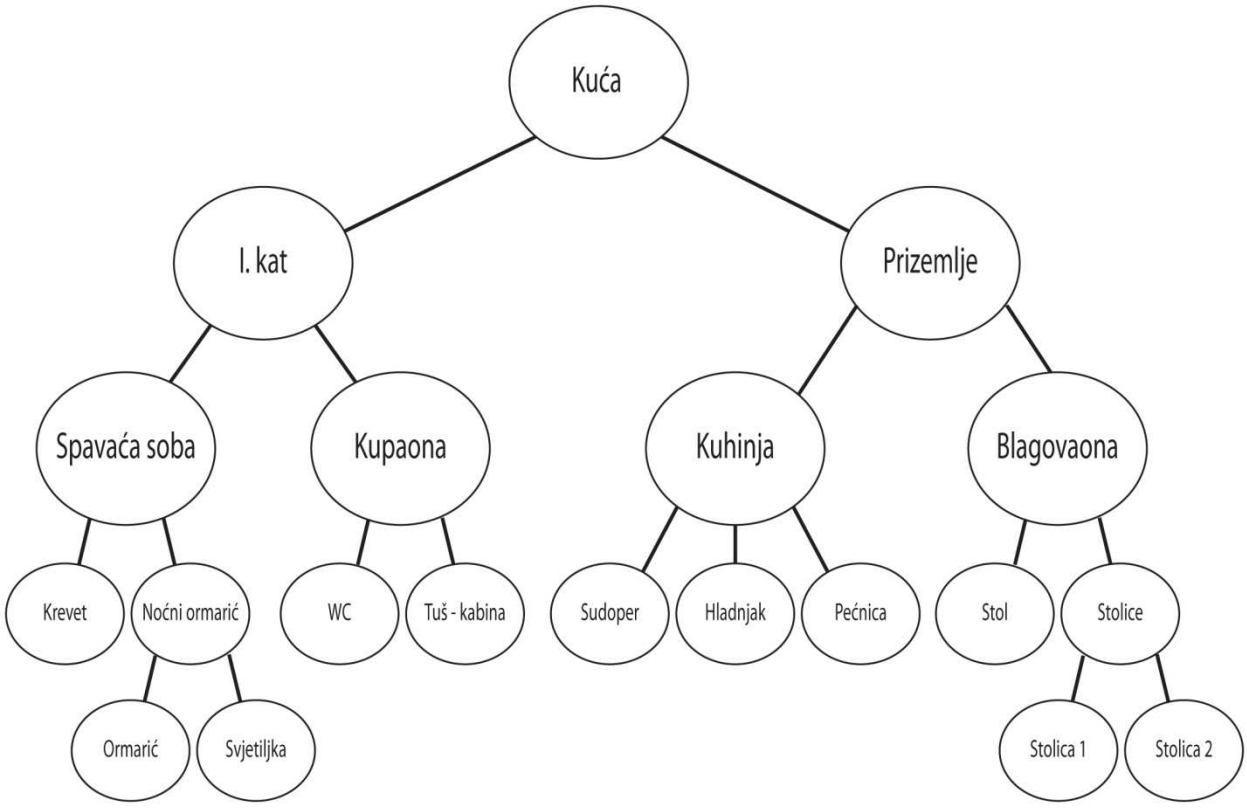
Postoje još neke vrste čvorova kao što su LOD (Level Of Detail) ili Switch čvorovi koji se koriste za određivanje razine detalja i uključivanje/isključivanje čvorova.

Ono što svakako treba spomenuti su prolasci kroz graf. To su rekurzivne funkcije koje, kao argument, dobivaju korijen grafa scene. U OpenSceneGraph-u, kao i u mojoj implementaciji, postoje tri prolaska kroz graf.

Update prolazak – dozvoljava aplikaciji da promijeni graf scene. Kod ovog prolaska se npr. mogu mijenjati pozicije predmeta. Predmeti se mogu rotirati, translatirati i skalirati po svim osima. U ovom prolasku se također osvježavaju i obujmice predmeta koje su orijentirane po osima koordinatnog sustava.

Cull prolazak – tijekom ovog prolaska testira se da li su objekti u sceni ili izvan nje. Tu se primjenjuju algoritmi koje ću kasnije objasniti u radu. Ovdje je još važno napomenuti da se kod ovog prolaska gradi lista za iscrtavanje koja se kasnije šalje na ostvarivanje prikaza.

Draw prolazak – iterira kroz listu stvorenu u prethodnom prolasku i iscrtava objekte nisko razinskim OpenGL API pozivima. Konkretno glDrawElements funkcija.



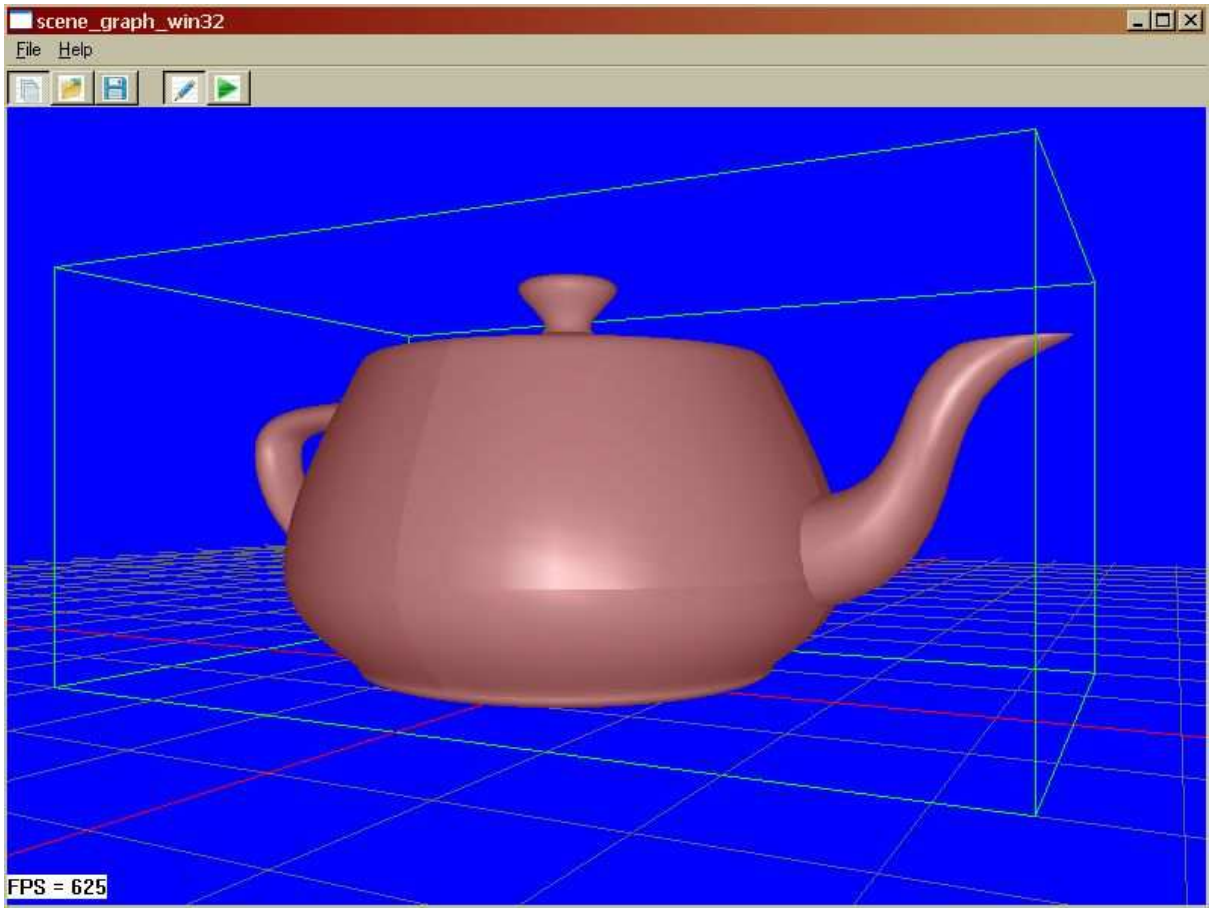
Slika 2. Primjer grafa scene.

Na slici 2 je jedan grubi primjer grafa scene na kojem se vidi logička organizacija scene kao i prednosti kod prostornih transformacija.

2.2. Graf scene i optimizacija iscrtavanja

Struktura grafa scene se može iskoristiti i za odbacivanje suvišnih poligona što je jedna od glavnih optimizacija iscrtavanja.

Ovdje bi trebali pojasniti pojam obujmica. Obujmice su jednostavna geometrijska tijela koja predstavljaju složene geometrijske oblike. To mogu biti sfere, kvadri paralelni s osima, općeniti kvadri, konveksne ljuske... Obujmice aproksimiraju predmet radi što bolje manipulacije. Na slici 3 je primjer iz moje aplikacije na kojoj se jasno vidi aproksimacija čajnika u obliku kvadra paralelnog s koordinatnim osima (engl. axis aligned bounding box).



Slika 3. Primjer obujmice za čajnik.

Uzmimo npr. odbacivanje poligona po projekcijskom volumenu (View Frustum Culling). Glavna ideja jest odbaciti sve poligone koji nisu u vidokrugu kamere. Uzeli bi parametre od projekcijskog volumena, položaja kamere i usmjerenja kamere i zatim bi krenuli od korijena grafa scene i ispitali bi da li je obujmica korijena scene u projekcijskom volumenu. Ako nije, odbacili bi cijelu scenu. Ako je, onda bi isto napravili za djecu toga čvora.

Odmah se može uočiti prednost grafa scene kod ovog algoritma. Vratimo se na sliku 2 i pretpostavimo da se trenutno nalazimo u kuhinji. To znači da smo u prizemlju i lako možemo odbaciti cijeli podgraf prvoga kata. Ako je usmjerenje kamere takvo da projekcijski volumen ne sadrži obujmicu blagovaone, onda i taj podgraf možemo odbaciti. No ako sadrži blagovanu, krećemo sa daljnjim ispitivanjem djece toga čvora.

Premda je ovaj jednostavan algoritam odličan i sigurno će ubrzati iscrtavanje, javlja se nekoliko problema.

Ako imamo veliki svijet s kompleksnim objektima, ovaj algoritam neće biti dovoljan i rezultati optimizacije će biti neprimjetni. Recimo da vidimo samo mali dio kompleksnog objekta, mi ćemo taj cijeli objekt iscrtati. U scenama vanjskog prostora često je slučaj da je cijela ili većina scene u projekcijskom volumenu. Zaklonjeni poligoni će isto biti iscrtani.

Upravljanje objektima u hijerarhijskom grafu scene može se poboljšati uvođenjem specijalnih struktura za prostorno dijeljenje kao što su BSP stablo ili oktalno stablo.

Tehnika razine detalja (engl. Level Of Detail) također pridonosi optimizaciji iscrtavanja scene i, premda nije tema ovog rada, treba se spomenuti u ovom kontekstu. Algoritmi ove tehnike računaju udaljenost između kamere i predmeta i, ovisno o toj udaljenosti, određuje broj poligona od kojih će se graditi predmet. Pretpostavlja se , ako je predmet dovoljno udaljen, da se razlika neće primijetiti. [1]

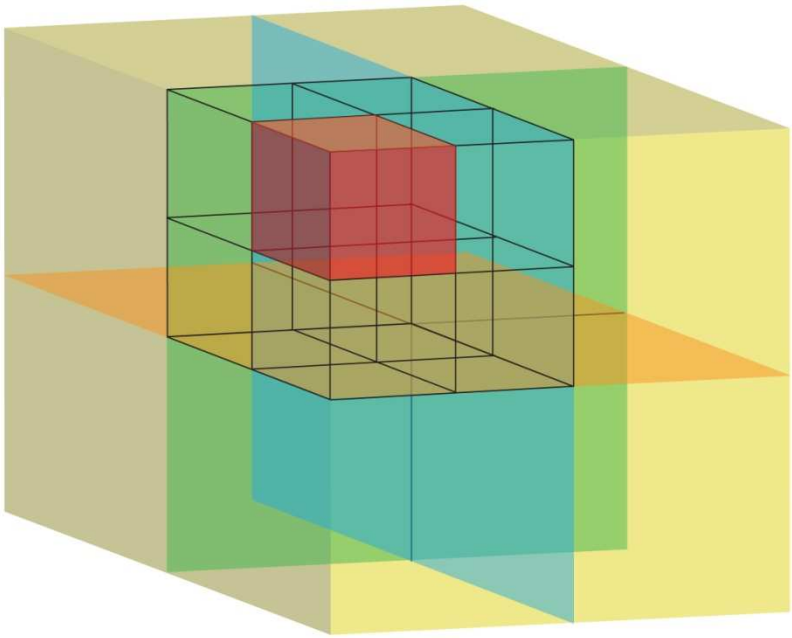
3. Strukture za prostornu podjelu scene

Razne prostorne strukture se koriste radi olakšanja manipulacije objektima i njihovog odbacivanja. Eve neke najvažnije...

3.1. Oktalno stablo

Oktalno stablo je struktura za podjelu prostora. Korijen tog stabla je kvadar koji se dijeli na osam dijelova. Svaki od tih dijelova se dalje može rekurzivno dijeliti na osam dijelova. Kvadar dijelimo stavljajući točku u sredinu koju koristimo kao vrh idućih osam manjih kvadrova. Maksimalni broj poligona koji želimo u svakom listu ili prazan čvor nam je uvjet kraja rekurzije. Time dobivamo objekt/scenu podijeljenu u proizvoljno velike "kutije". [3]

Oktalno stablo je iznimno popularna struktura, pogotovo kad govorimo o odbacivanju suvišnih poligona. Na slici 4 vidimo vizualan primjer takve strukture.

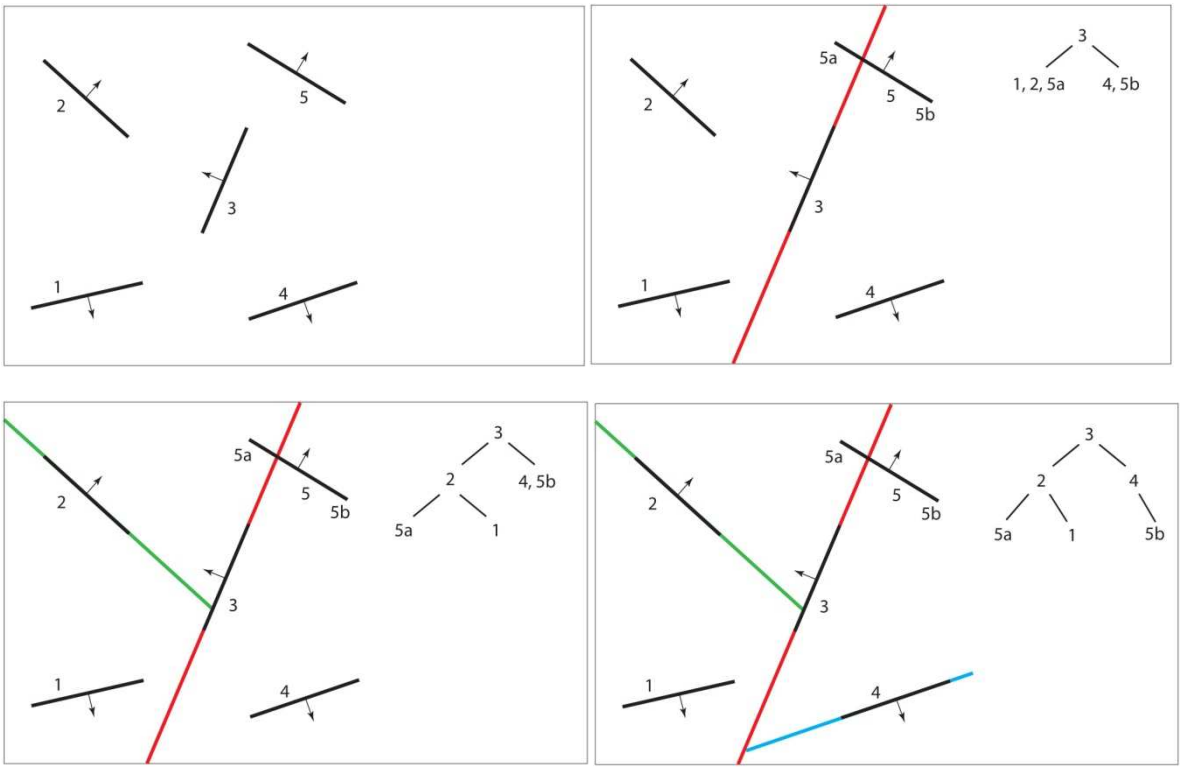


Slika 4. Izgled oktalnog stabla u prostoru.

3.2. BSP stablo

Kratica BSP stoji za Binary Space Partitioning. Stablo se gradi na taj način da se izabere jedan poligon u sceni i postavi ga se u korjen grafa. Scena se podijeli na poligone ispred i iza izabranog poligona. Oni poligoni koji su iza tvore desnu stranu grafa, a lijevu tvore oni koji su ispred. Opet se odabire jedan poligon i postupak se ponavlja.

Na slici 5 je kopija ilustracija iz knjige Igora S. Pandžića, a zorno predstavlja ideju BSP algoritma. [1]



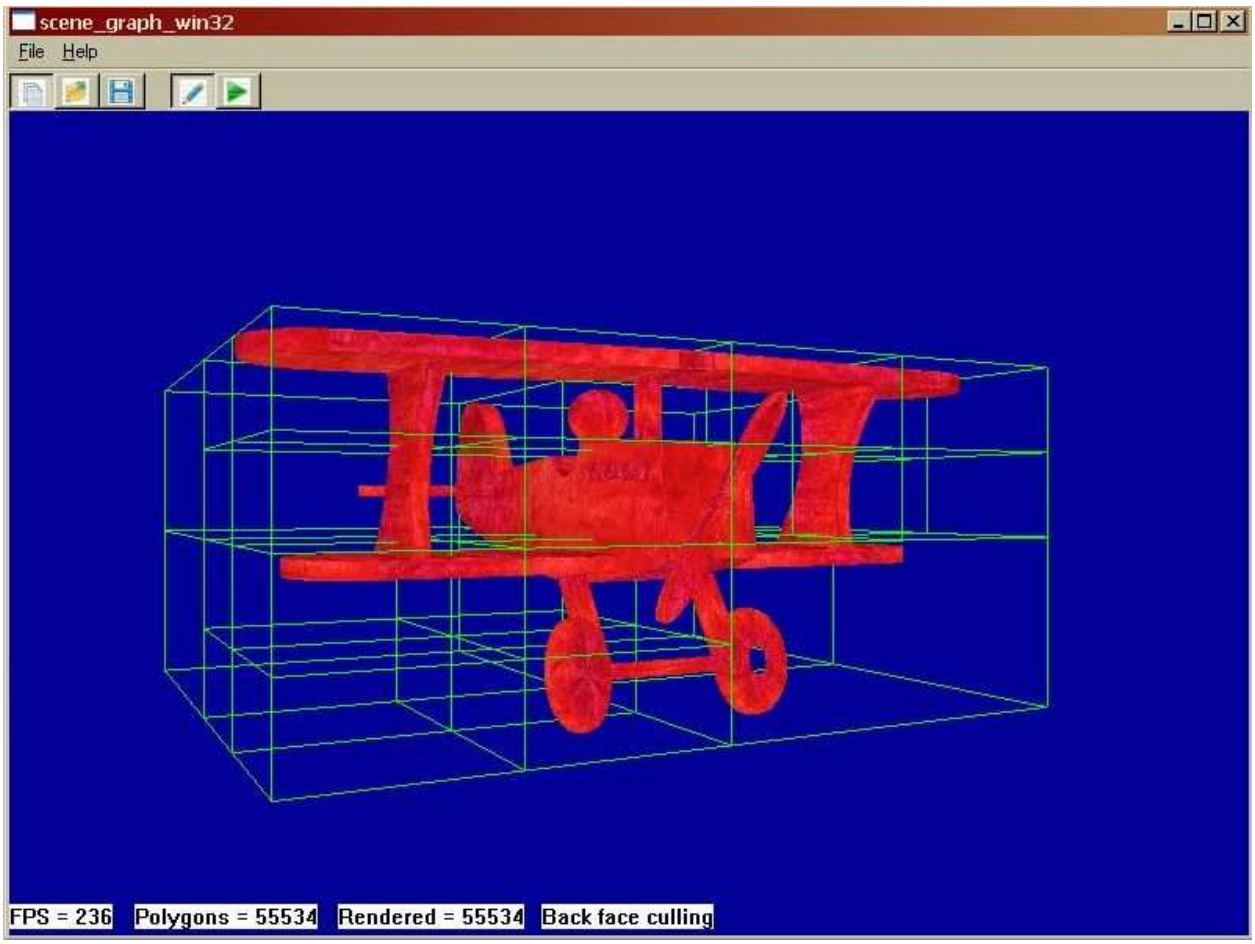
Slika 5. Algoritam stvaranja BSP stabla.

3.3. Kd-stablo

KD-stablo (k-dimensional tree) je specijalni slučaj BSP stabla. Svaki čvor koji nije list dijeli prostor na dva dijela. Sve točke lijevo od te ravnine, nalaze se u lijevom dijelu grafa, a sve točke desno, nalaze se u desnom dijelu grafa. Ravnina se određuje tako da se svaki čvor poveže s jednom od K dimenzija. Ako je za određenu podjelu korištena X os, onda će se sve točke razvrstati po svojoj X koordinati. [4]

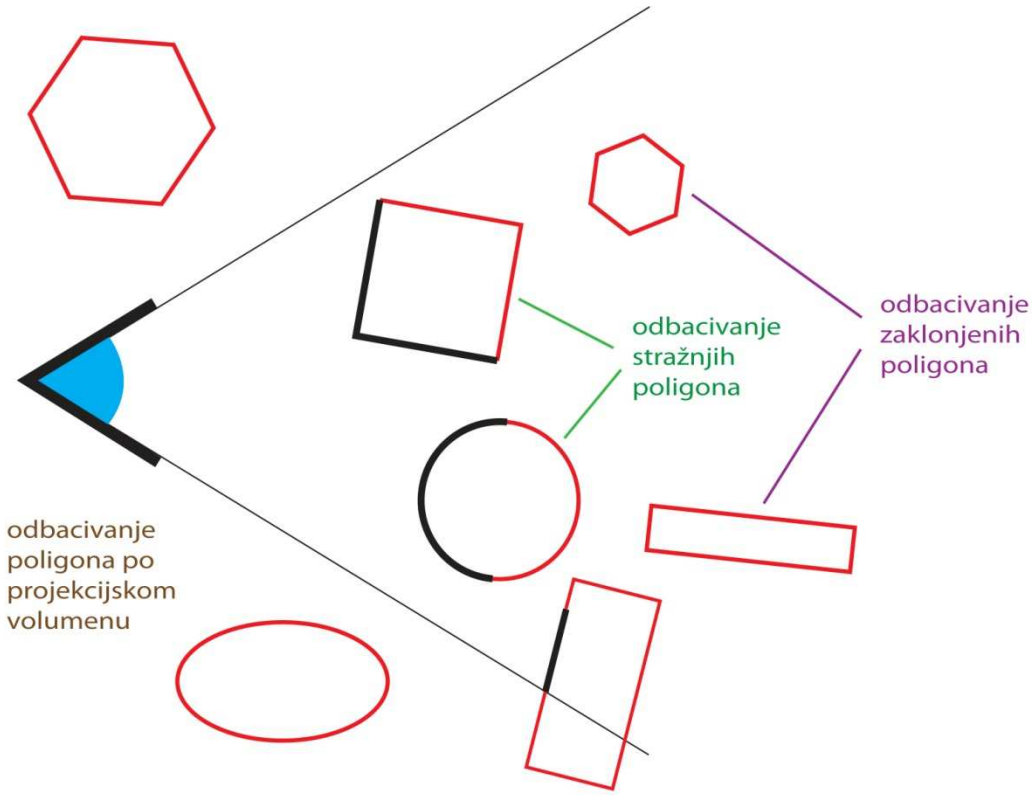
3.4 Integracija grafa scene i oktalnog stabla

U svojoj implementaciji koristim kombinaciju grafa scene i oktalnog stabla za prostornu podjelu. Cijela scena je podjeljena u graf scene, a obujmice listova su korijeni oktalnog stabla. Oktalno stablo gradim samo kod kompleksnijih objekata, a uvjet daljnje razdiobe mi je broj vrhova u pojedinom čvoru stabla. Taj uvjet iznosi 5% od ukupnog broja vrhova. Na slici 6 stoji primjer kompleksnog objekta podijeljenog u oktalno stablo. Ovakvim načinom podijele se izbjegava iscrtavanje čitavog objekta ako je samo dio vidljiv. [3]



Slika 6. Primjer podijele kompleksnog objekta u oktalno stablo.

4. Metode uklanjanja poligona



Slika 7. Podijela algoritama za odbacivanje suvišnih poligona.

4.1. Odbacivanje stražnjih poligona

Ovo je osnovni oblik odbacivanja suvišnih poligona i dio je geometrijske faze grafičkog protočnog sustava. Ako normale nisu eksplicitno zadane u modelu, onda se određuju prema redoslijedu vrhova. Redoslijed vrhova (normala površine) nam odaje orijentaciju i omogućava odbacivanje onog poligona koji je okrenut prema natrag ovisno o kutu gledanja.

Odbacivanje stražnjih poligona automatski obavlja OpenGL API jednom naredbom uključivanja te funkcije. Ova metoda odbacuje otprilike pola poligona u sceni pri čemu se dobivaju velike uštede.

4.2. Odbacivanje poligona po projekcijskom volumenu

Algoritmi za odbacivanje poligona po projekcijskom volumenu se oslanjaju na činjenicu da je, ono što nije u vidokrugu kamere, nevidljivo tj. nepotrebno iscrtavati.

Projekcijski volumen je krnja piramida čiji vrh čini bližu, a baza dalju graničnu plohu. Ponekad se, za vrijednost dalje granične plohe, stavlja beskonačna vrijednost tako da se iscrta sve što je u vidokrugu. U tim slučajevima je tehnika razine detalja jako korisna.

Kada bi za svaki poligon u sceni radili proračune da li je u projekcijskom volumenu ili ne, onda bi bolje prošli da jednostavno sve poligone pošaljemo u protočni sustav. Iz ovog razloga koristimo obujmice i strukture za podjelu prostora.

U svojoj implementaciji koristim malo izmijenjen kod koji nude na internetskoj stranici http://www.lighthouse3d.com/opengl/viewfrustum/ kao primjer učenja algoritma. Tu postoji razred Frustum koji predstavlja krnju piramidu tj. projekcijski volumen koji se sastoji od 6 ravnina. Za svaku ravninu se izračunaju normale. Ja sam izmijenio funkciju pointInFrustum koja ispituje gdje se nalazi točka u odnosu na površinu kojoj je orijentacija određena normalom. Za svaku površinu ispitujem svih osam točaka obujmice čvora. Ako se svih osam nalazi na krivoj strani jedne površine, onda je čvor izvan projekcijskog volumena. Ako se svih osam točaka nalazi na unutrašnjoj strani svih šest ravnina, onda je čvor unutar projekcijskog volumena. Za sve ostalo funkcija vraća vrijednost presijecanja projekcijskog volumena. [5]

4.3. Odbacivanje zaklonjenih poligona

Nakon odbacivanja stražnjih poligona i poligona koji nisu u projekcijskom volumenu, još uvijek imamo velik broj zaklonjenih poligona koje nije potrebno iscrtavati. Algoritmi za odbacivanje zaklonjenih poligona se brinu o tom problemu.

Ove algoritme dijelimo na one koji odbacuju poligone u odnosu na točku gledanja i one koje to rade ovisno o dijelu scene (regije) u kojem se promatrač nalazi.

4.3.1. Algoritmi regije

Algoritmi regije računaju vidljivost iz svih dijelova mape. Gdje god se promatrač nalazi, aplikacija ima listu čvorova potencijalno vidljivih iz te pozicije (PVS – Potentially Visible Set). To se računa u fazi pretprocesiranja i stoga ovi algoritmi nisu povoljni za dinamičke scene.

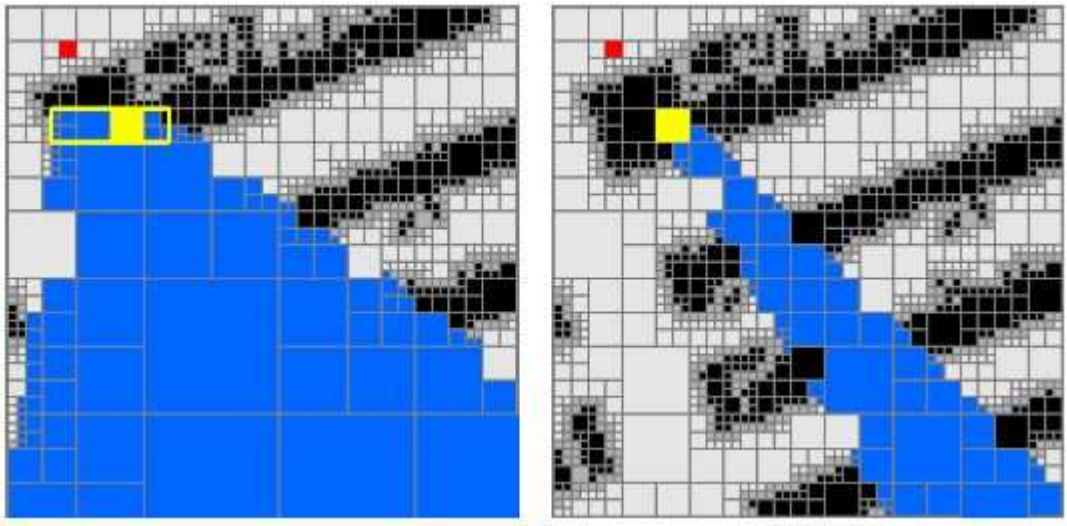
Svi algoritmi odbacivanja zaklonjenih poligona bazirani na ćelijama pretpostavljaju da se virtualni svijet može podijeliti na ćelije koje su spojene portalima. Ako se portal ne vidi, iz određenog položaja, možemo odbaciti sve poligone u toj ćeliji.

Danas se najviše koriste dvije metode. Jedna bazirana na BSP-u (Binary Space Partitioning), a druga na portalnom odbacivanju.

Seth Teller je napravio zapažen algoritam kada je riječ o ćelijama i BSP-u [6][7]. Prvo se scena podijeli u konveksne ćelije koristeći BSP. Prozirni dijelovi ćelije, kao što su portali, se pronalaze na granicama ćelije. Vidljivost od ćelije do ćelije se testira linijama poveznicama. Ako postoji linija koja spaja dvije ćelije, jasno je da ide kroz portal. Dakle, samo moramo odrediti vidljivost portala u jednoj ćeliji iz druge. Tada se gradi graf koji pokazuje koje ćelije su susjedne. Po tome grafu se gradi PVS. To se radi za svaku ćeliju u sceni. Sličan algoritam se koristi u igri Quake.

Ovakvi su algoritmi odlični za prikaz unutrašnjeg prostora i tu se najviše i koriste.

Schaufler je napravio algoritam [8] koji dijeli prostor u oktalno stablo koje ima prazne, granične i neprozirne čvorove. Nađu se neprozirni čvorovi i, za svaki čvor, se odredi koliki dio prostora taj čvor zaklanja iz svake pozicije čvora iz kojeg se gleda. Po tome se gradi PVS.



Slika 7. Primjer iz Schauflerovog rada. Crveno je obojana ćelija iz koje se gleda (engl. viewcell), žuto predstavlja neprozirni čvor koji se proširuje na susjedne čvorove ako su i oni neprozirni. Treba napomenuti da se, svi prazni čvorovi koji se nalaze unutar graničnih, klasificiraju kao neprozirni. Plavom bojom označen je zaklonjen prostor.

4.3.2. Algoritmi točke gledanja

Algoritmi točke gledanja računaju vidljive poligone on-line. Pogodni su za dinamičke scene jer nemaju unaprijed određeni PVS.

Kod algoritama točke gledanja također postoji podjela na ćelije. Luebke i Georges su osmislili algoritam [9] koji prvo iscrta ćeliju u kojoj se promatrač nalazi i identificiraju se portali koji su u projekcijskom volumenu. One ćelije, koje se vide iz portala, se rekurzivno ponovo ispituju s time da se kut sve više i više smanjuje, ovisno o veličini portala. Mana ovakvih algoritama jest ta što su izračuni prilično skupi, no prekomjernog iscrtavanja nema. Dobra strana je da nema potrebe za pretprocesiranjem i ne postoji PVS što znači da je pogodan za dinamične scene.

Hudson je osmislio algoritam [10] baziran na volumenu sjene. Dijeli objekte u sceni na zaklonjene i zaklanjajuće. Pretpostavlja da se zaklonjeni predmet neće vidjeti ako se promatrač nalazi u sjeni zaklanjajućeg objekta. Zaklanjajući objekti se unaprijed određuju. Za svaki od najboljih n zaklanjajućih objekata, koji se nalaze u projekcijskom volumenu, gradi se volumen sjene (krnja piramida) ovisno o točki gledanja i siluete zaklanjajućeg objekta. Hijerarhija scene testira se sa svakom od sjena. Ako je čvor potpuno zaklonjen sjenom,

odbacuje se. Ako je čvor potpuno vidljiv, svi čvorovi ispod njega se iscrtavaju. Ako je parcijalno vidljiv, provode se testovi njegove djece (rekurzivno).

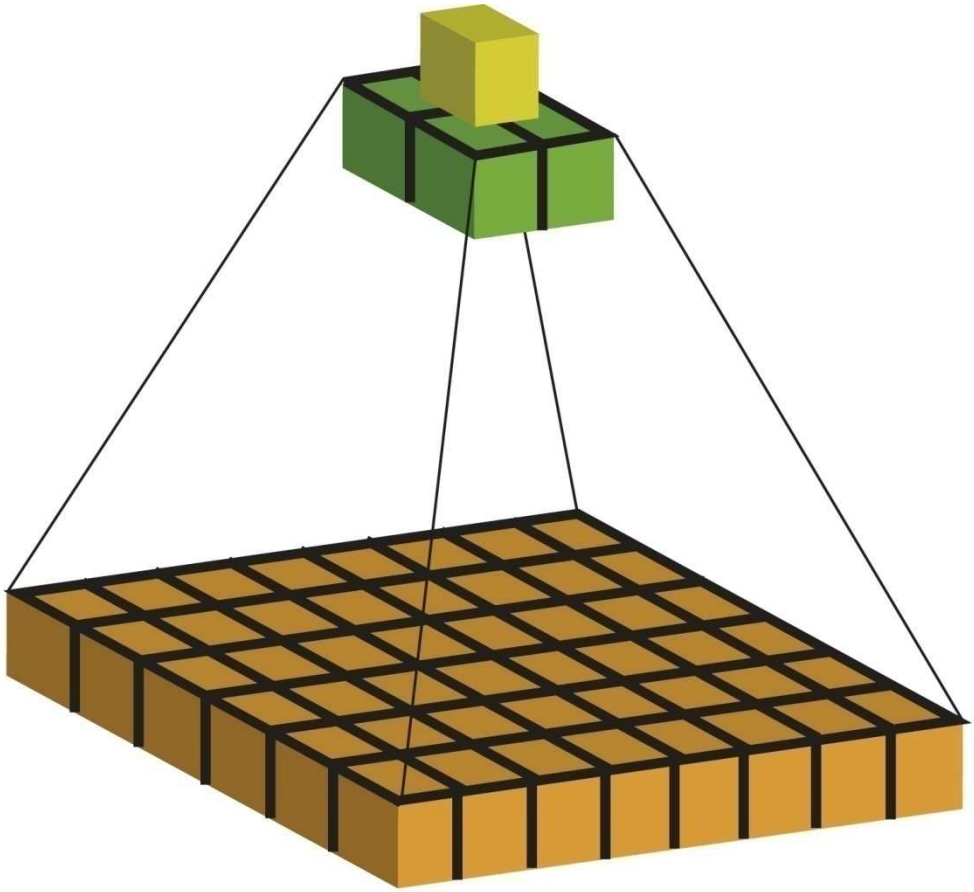
Ovaj algoritam poboljšao je Bittner [11] kombinirajući BSP stablo s volumenom sjene.

Još jedan način utvrđivanja zaklonjenih poligona temelji se na odašiljanju zrake (ray casting). Zraka se odašilje za svaki slikovni element iz točke gledišta. Prvi objekt kojeg presijeca zraka određuje slikovni element koji će se iscrtati. Ovaj algoritam nikad neće iscrtati nešto što nije u sceni.

4.3.3. Odbacivanje pomoću sklopovlja

Kada govorimo o odbacivanju poligona pomoću sklopovlja, prvo naiđemo na pojam Z-spremnika. Kada neki objekt iscrtamo na ekran pomoću grafičke kartice, dubina iscrtanog slikovnog elementa se spremi u Z-spremnik. Ako se neki drugi objekt mora iscrtati u istom slikovnom elementu, Z-spremnik će usporediti dubinu novog slikovnog elementa s dubinom spremljenom u Z-spremniku i iscrtat će onog bližeg promatraču, a odbaciti onog daljeg. Bliža vrijednost se sprema u Z-spremnik. Ovo odbacivanje se događa u rasterskoj fazi grafičkog protočnog sustava.

Ned Greene je osmislio hijerarhijski Z-spremnik algoritam [12]. On koristi oktalno stablo za dijeljenje prostora, a Z-piramidu za utvrđivanje vidljivosti čvora.



Slika 8. Z-piramida je struktura čija je najmanja razina (baza) vrijednost Z-spremnika. Svaki idući nivo sadrži najudaljeniju Z vrijednost od četiri Z-a (2x2). Završni, tj. najviši nivo sadrži samo jednu vrijednost, a to je najudaljeniji Z od promatrača na cijeloj slici. Na slici narančasti dio predstavlja Z-spremnik, a žuta kockica na vrhu predstavlja najudaljeniju vrijednost.

Da li je čvor vidljiv ili ne određujemo tako da uzmemo prednju stranu čvora (obujmice) i uspoređujemo ju sa Z-piramidom. Ako je najbliža točka prednje strane dalja od vrha piramide, možemo odbaciti cijeli čvor. Ako to nije slučaj, onda se rekurzivno spuštamo niz piramidu i u svakom koraku napravimo istu dubinsku usporedbu. U slučaju da na kraju ne nađemo nijedan vidljivi dio obujmice, možemo odbaciti čvor.

Greene u svojem algoritmu koristi i listu čvorova baziranu na vremenskoj koherenciji tj. pretpostavlja da će, čvorovi koji su bili vidljivi u prošlom trenutku prikaza, biti vidljivi i u idućem.

Postoji još jedna mogućnost kada se radi o sklopovskoj pomoći pri odbacivanju poligona, a to su sklopovski upiti o zaklanjanju (engl. hardware occlusion queries).

Premda se oba načina oslanjaju na sklopovlje, postoji velika razlika između odbacivanja pomoću Z-spremnika i upita o zaklanjanju. Odbacivanje pomoću upita o zaklanjanju radi u geometrijskoj fazi protočnog sustava, dok Z-spremnik odbacuje poligone u rasterskoj fazi.

Oba načina zahtijevaju od programera da sortira objekte od naprijed prema natrag (front-to-back). Kod odbacivanja pomoću upita o zaklanjanju postoji problem usklađivanja centralnog procesora i grafičkog procesora. Normalan način je da CPU i GPU rade paralelno. Kada CPU izda naredbu GPU-u, ne čeka da ovaj završi nego se naredbe spremaju u red što dozvoljava glavnom procesoru da nastavi s radom. GPU izvršava naredbe kada je spreman. Kod upita o zaklanjanju GPU treba reći CPU-u koliko slikovnih elemenata je iscrtano na zaslon što znači da ih prvo treba iscrtati. Sada CPU mora čekati dok GPU ne završi sa iscrtavanjem. GPU, kod upita o zaklanjanju, ne iscrtava ništa na zaslon jer je iscrtavanje onesposobljeno, no vraća broj vidljivih slikovnih elemenata tj. onih koji bi bili vidljivi da su iscrtani na zaslon.

Zbog ovog problema su nastali algoritmi CHC (engl. Coherent Hierarchical Culling) [13], NOHC (engl. Near Optimal Hierachical Culling) [14], CHC++ [15]... Problem su riješili koristeći više upita u isto vrijeme (engl. multiquering), vremensku koherenciju kod slanja upita grafičkoj kartici i napredan način izgradnje PVS-a.

5. Implementacija algoritama za odbacivanje zaklonjenih poligona

U ovom radu odlučio sam proučiti detaljnije algoritme za odbacivanje zaklonjenih poligona pomoću sklopovskih upita o zaklanjanju. Temelj ovih algoritama su upiti upućeni grafičkoj kartici, no veliki problem čini već spomenuta asinkrona priroda rada ovih upita. Da bih to bolje razumio prvo sam implementirao naivnu "stani i čekaj" (eng. stop and wait) metodu.

5.1 Hijerarhijska "stani i čekaj" metoda

Ova metoda je dosta jednostavna, a funkcionira tako da, za svaki čvor u hijerarhiji, postavi upit da li je, objekt koji želimo iscrtati, zaklonjen ili ne. Ako je zaklonjen, preskačemo taj čvor. Ako nije zaklonjen, djecu toga čvora stavljamo u red uređen po udaljenosti od kamere (od naprijed prema natrag, engl. front-to-back order). Ako je čvor list, onda ga iscrtamo. Dakle CPU pošalje upit i čeka rezultat upita i pritom ne radi ništa. Odmah se može pretpostaviti koja je loša strana ovog algoritma. Tek kad je velika većina scene zaklonjena, ovaj algoritam postaje koristan, no i onda kaska za naprednijim riješenjima. Treba napomenuti da se za potrebe upita iscrtavaju obujmice objekata, a ne sami objekti.

```
Stop and wait  begin
    DistanceQueue.push(Root);
    while !DistanceQueue.Empty() do
        if InsideViewFrustum(N) then
            IssueQuery(N);
            if IsVisible(N) then
                TraverseNode(N);
end
```

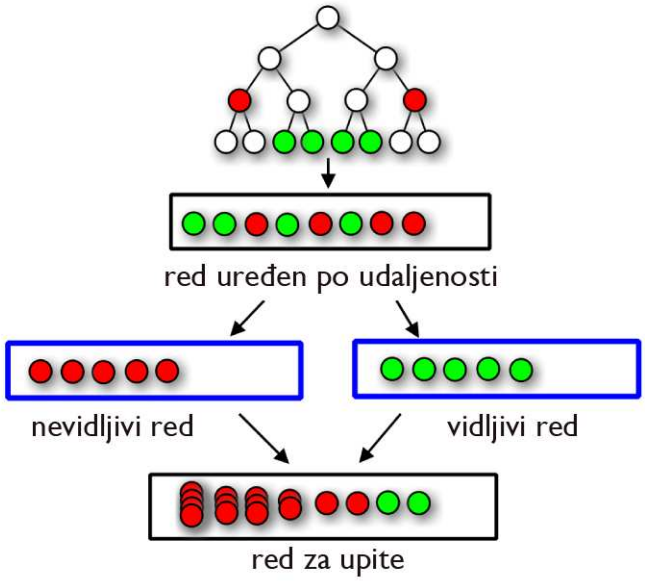
Ispis 1. Pseudokod "stani i čekaj" algoritma.

5.2 CHC++ algoritam

Oliver Mattausch, Jiri Bittner i Michael Wimmer osmislili su CHC++ algoritam. Za potrebe ovog rada, stupio sam u kontakt sa Jiri Bittnerom koji mi je posalo izvorni kod. Ja sam ga onda modificirao da radi sa mojim grafom scene. Algoritam riješava mnoge probleme vezane uz ovaj način odbacivanja zaklonjenih poligona.

Ovaj algoritam smanjuje broj promjena stanja i upita. Naišao sam na podatak da je oko 200 promjena stanja, u jednom trenutku prikaza, prihvatljivo za današnje sklopovlje. Kod ovakvih algoritama to se može premašiti za više puta. CHC++ uspijeva napraviti redukciju tako da radi jedan upit za više čvorova, a ne samo za jedan čvor. Također smanjuje i čekanje glavnog procesora. Dok se čekaju odgovori na upite postavljene grafičkoj kartici, ovaj algoritam šalje upite za čvorove koji su bili vidljivi u prošlom trenutku prikaza i dalje prolazi kroz hijerarhiju scene.

U CHC++ algoritmu se isto koristi red uređen po udaljenosti od kamere, no također se bilježe i podatci o pojedinim čvorovima. Za svaki čvor se zna koliko je puta do sada bio nevidljiv, u kojem trenutku prikaza je zadnji put bio ispitan i, ako je vidljiv, koliko se još očekuje da ostane vidljiv.



Slika 9. Grafički prikaz raspodjele čvorova po redovima [15].

Vidljivi listovi se odmah iscrtavaju i stavljaju se u red vidljivih čvorova za upit o zaklanjanju, no odgovor na taj upit se koristi tek u idućem trenutku prikaza.

Nevidljivi čvorovi se stavljaju u red za upit nevidljivih čvorova. Kada broj čvorova, u tom redu, dosegne određenu vrijednost, pošalje se jedan upit za više čvorova (engl. multiquery).

Odluka o tome koji čvorovi će biti grupirani za jedan upit ovisi o odnosu broja čvorova u potencijalnom upitu i očekivanog troška upita:

$$V(Mj) = \frac{B(Mj)}{C(Mj)}$$

Gdje je B broj upita, a C funkcija očekivanog troška definirana formulom:

$$C(M) = 1 + p_{vidljiv}(M) * |M|$$

$|M|$ predstavlja broj čvorova u upitu, broj 1 je trošak samog upita, a $p_{vidljiv}(M)$ je vjerojatnost da rezultat upita bude vidljiv. U tom slučaju moramo sve čvorove pojedinačno ispitati.

$$p_{vidljiv}(M) = 1 - \prod_{\forall N \in M} p_{isto\_stanje}(i_N)$$

p_{isto_stanje} je vjerojatnost da čvor ostane u istom stanju, a definirana je formulom:

$$p_{isto\_stanje}(i) \approx 0.99 - 0.7e^{-i}$$

i predstavlja koliko je puta čvor bio nevidljiv. Ovo je analitička formula koju su dobili autori algoritma testirajući ga na različitim scenama.

Ako je čvor bio nevidljiv, a postane vidljiv u jednom trenutku prikaza, za njega se izračuna slučajna vrijednost koja se zbroji sa trenutnim brojem trenutka prikaza i taj broj nam predstavlja trenutak prikaza do kojeg se pretpostavlja da će ovaj čvor biti vidljiv. Taj će čvor biti ponovo ispitan najkasnije za 10 trenutka prikaza. Također ako je list vidljiv, onda sve

njegove pretke učinimo vidljivima. To nam pomaže da, u idućem trenutku priakaza, ne ispitujemo uzalud unutarnje čvorove hijerarhije.

```
CHC++ begin
    DistanceQueue.push(Root);
    while !DistanceQueue.Empty() || !QueryQueue.Empty() do
        while !QueryQueue.Empty() do
            if FirstQueryFinished then
                N = QueryQueue.Dequeue();
                HandleReturnedQuery(N);
            else
                // next prev. vis. node query;
                IssueQuery(v-queue.pop());
        if !DistanceQueue.Empty() then
            N = DistanceQueue.DeQueue();
            if InsideViewFrustum(N) then
                if !WasVisible(N) then
                    QueryPreviouslyInvisibleNode(N);
                else
                    if N.IsLeaf && QueryReasonable(N)
                    then
                        v-queue.push(N);
                    TraverseNode(N);
        if DistanceQueue.Empty() then
            // issue remaining query batch;
            IssueMultiQueries();
    while !v-queue.empty() do
        // remaining prev. visible node queries;
        IssueQuery(v-queue.pop());
end
```

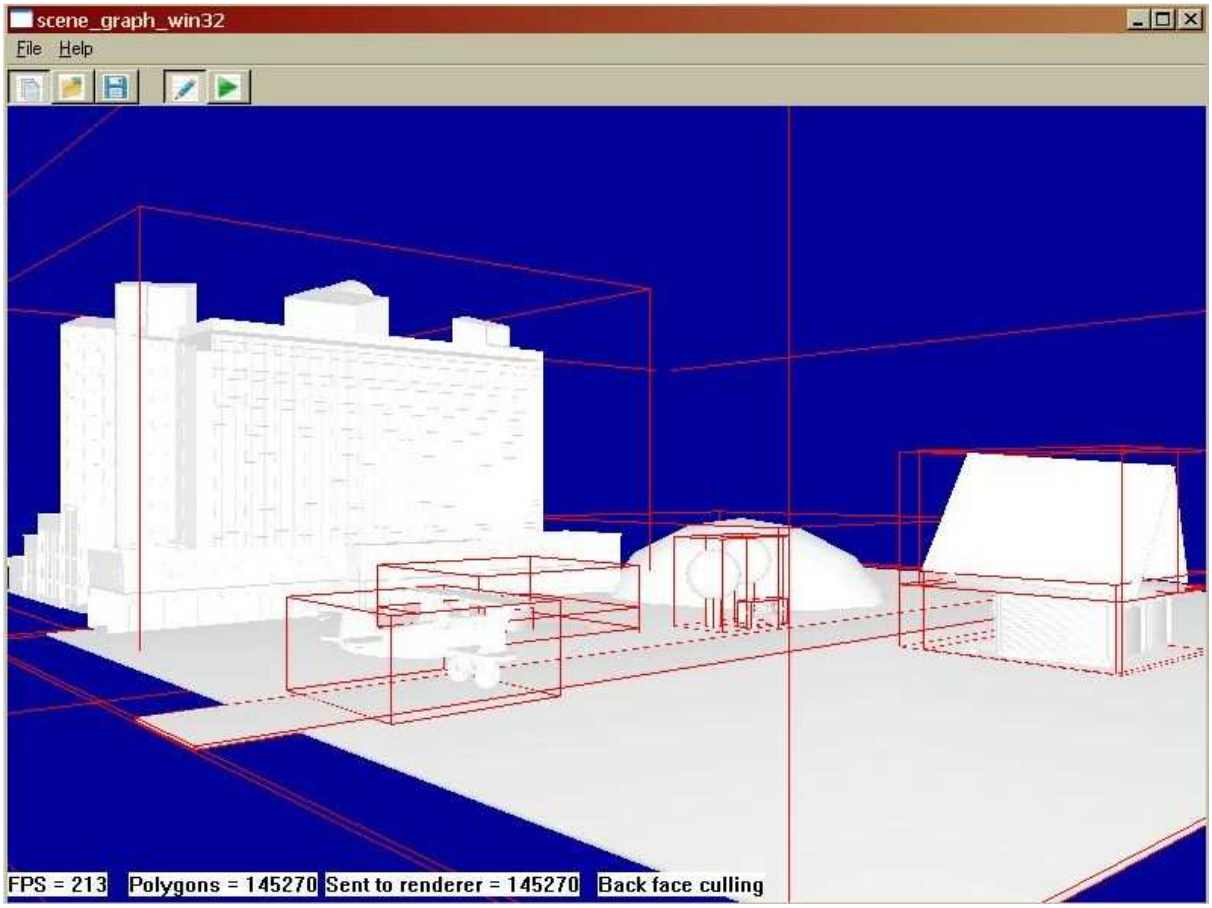
```
TraverseNode(N) begin
    if IsLeaf(N) then
        Render(N);
    else
        DistanceQueue.PushChildren(N);
        N.IsVisible = false;
end
PullUpVisibility(N) begin
    while !N.IsVisible do
        N.IsVisible = true; N = N.Parent;
end
HandleReturnedQuery(Q) begin
    if Q.visiblePixels > threshold then
        if Q.size() > 1 then
            QueryIvididualNodes(Q); // failed multiquery
        else
            if !WasVisible(N) then
                TraverseNode(N);
            PullUpVisibility(N);
    else
        N.IsVisible = false;
end
QueryPreviouslyInvisibleNode(N) begin
    i-queue.push(N) ;
    if i-queue.size() ≥ maxPrevInvisNodesBatchSize then
        IssueMultiQueries(); // issue the query batch

IssueMultiQueries() begin
    while !i-queue.Empty() do
        MQ = i-queue.GetNextMultiQuery();
        IssueQuery(MQ); i-queue.PopNodes(MQ);
end
```

Ispis 2. Pseudokod CHC++ algoritma.

6. Rezultati implementiranih optimizacija

Za mjerenje rezultata optimizacije sam složio virtualnu scenu u vlastitom editoru. Neke objekte sam oblikovao, a neke sam učitao iz .obj formata i uvrstio ih u scenu. Scena se sastoji od 145270 poligona. Za algoritam odbacivanja poligona po projekcijskom volumenu je vrlo bitna struktura tj. graf scene. Nastojao sam grupirati dijelove scene imajući na umu logičku i prostornu organizaciju. Na slici 10 se vidi cijela scena, bez tekstura i sa iscrtanim obujmicama svih čvorova grafa scene.



Slika 10. Pogled na cijelu scenu sa obujmicama.

Da scena ne bi bila tako monotona, dodao sam i teksture (Slika 11). Odmah možemo uočiti pad FPS-a. Razlog tog pada je jasan. Osim samih poligona sada je potrebno iscrtati i teksture. U oba slučaja koristim samo odbacivanje stražnjih poligona.

Slika 11. Pogled u cijelu scenu sa teksturama.

6.1 Odbacivanje stražnjih poligona i po projekcijskom volumenu

Ako sada uključim i odbacivanje po projekcijskom volumenu neću dobiti nikakvo ubrzanje niti usporavanje iscrtavanja. Ubrzanje neću dobiti jer ništa ne odbacujemo, čitava scena je u projekcijskom volumenu. Do usporavanja također neće doći jer se ispituju samo veći čvorovi grafa scene. Uzmimo za primjer model klupice i dva drveta. To je jedno podstablo čiji korijen sadrži obujmicu sva tri objekta i samo se ta obujmica ispituje. Funkcija za provjeru će vratiti vrijednost takvu da je cijela obujmica u projekcijskom volumenu i tu prestaje ispitivanje podstabla.

Oni slučajevi koji su najzanimljiviji za testiranje su oni kod kojih dolazi do ubrzanja ili usporavanja iscrtavanja. Promotrimo slučaj na slikama 12a do 12e.

Slika 12a. Isključeni svi algoritmi.

Slika 12b. Odbacivanje stražnjih poligona.

Slika 12c. Odbacivanje po projekcijskom volumenu.

Slika 12d. Odbacivanje stražnjih poligona i odbacivanje po projekcijskom volumenu.

Slika 12e. Odbacivanje po projekcijskom volumenu i odbacivanje stražnjih poligona sa malo izmijenjenim kutom gledanja.

Na slikama 12a – 12d prikazan je isti pogled u scenu s različitim optimizacijama iscrtavanja. U ovom pogledu scena se sastoji od dva stabla, klupe, ceste, brda, tla i vidi se mali komadić kompleksnog objekta (hotel) koji je podijeljen u oktalno stablo.

U prvom slučaju (slika 12a) sve šaljemo na iscrtavanje i ne odbacuju se stražnji poligoni. Dakle iscrtano je svih 145270 poligona. Broj FPS-a je oko 173.

U drugom slučaju (slika 12b) i dalje šaljemo sve poligone na iscrtavanje, no u geometrijskoj fazi se odbacuju stražnji poligoni. Rezultat je ubrzanje iscrtavanja, oko 206 FPS-a.

U trećem slučaju (slika 12c) sam isključio odbacivanje stražnjih poligona, a uključio odbacivanje po projekcijskom volumenu. Od 145270 na iscrtavanje je poslano 31549 poligona. Dolazi do ubrzanja iscrtavanja na 240 FPS-a.

Na tablici 1 preglednije su prikazani rezultati optimizacija sa slike 12.

Algoritam	Broj poligona poslanih na iscrtavanje	FPS
Bez algoritma (12a)	145270	173
Odbacivanje stražnjih poligona (12b)	145270	206
Odbacivanje po projekcijskom volumenu (12c)	31549	240
Odbacivanje stražnjih poligona + odbacivanje po projekcijskom volumenu (12d)	31549	304
Odbacivanje stražnjih poligona + odbacivanje po projekcijskom volumenu (12e)	1811	427

Tablica 1. Tablica uz sliku 12.

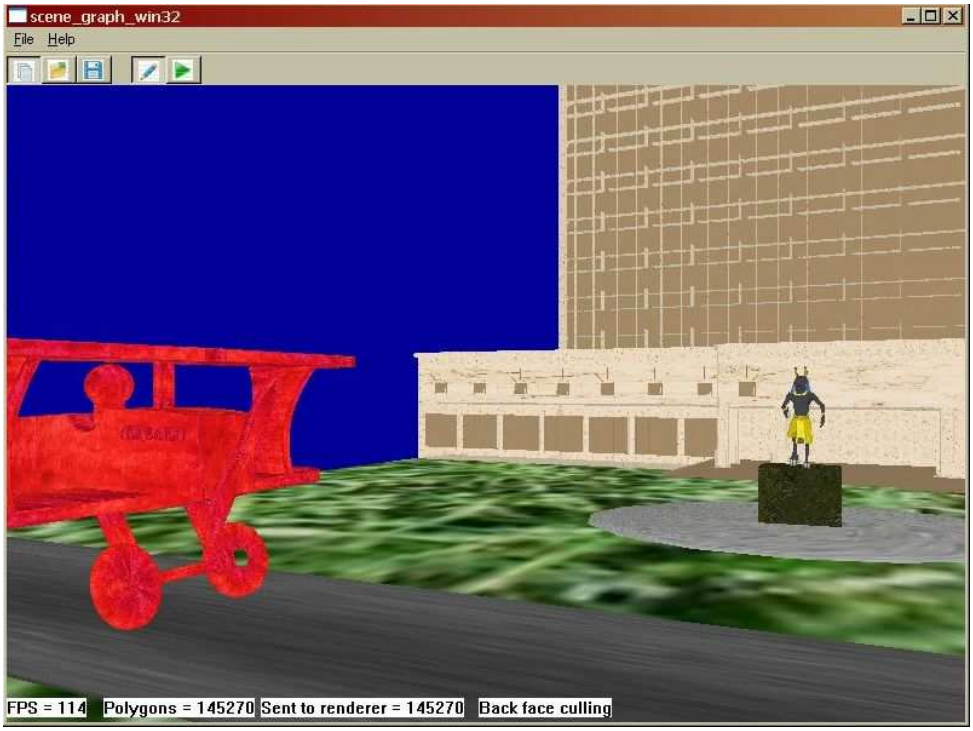
U četvrtom slučaju (slika 12d) je uključeno odbacivanje po projekcijskom volumenu i odbacivanje stražnjih poligona. Vidimo da dolazi do velikog ubrzanja na oko 304 FPS-a. Na ovom primjeru se izvrsno vidi efekt odbacivanja suvišnih poligona. Od slučaja kada smo isključili sve algoritme, dobili smo strahovito ubrzanje iscrtavanja od 173 do 304 FPS-a. Jasno je zbog čega dobivamo takve rezultate. U pogledu imamo dva čvora koja se kompletno nalaze u projekcijskom volumenu. Prvi čvor je brdo, a drugi dva stabla i klupa. Nadalje imamo dva čvora koje siječemo. Jedan je hotel, a drugi tlo i cesta. Tlo i cesta su jednostavni objekti i sastoje se od svega nekoliko poligona, pa njihovo ispitivanje i iscrtavanje možemo zanemariti. Ono što je nama interesantno jest hotel. Vidimo samo mali djelić hotela. Da nisam napravio dijeljenje kompleksnih objekata u oktalno stablo, morali bi, zbog tog malog vidljivog djelića, iscrtati cijeli hotel. Ovako funkcija vraća vrijednost presijecanja obujmice hotela i moramo ići u daljnje ispitivanje obujmica čvorova u oktalnom stablu. U ovom slučaju ispitivanje nije toliko kompleksno i kao rezultat dobivamo ubrzanje iscrtavanja. Ono što se

mora napomenuti jest da je hotel sačinjen od 68775 poligona. Vidimo na slici da mi šaljemo samo 31549 poligona skupa s ostatkom scene. I na kraju, najvažnija činjenica koja je dovela do ubrzanja jest ta da se većina scene nalazi izvan projekcijskog volumena.

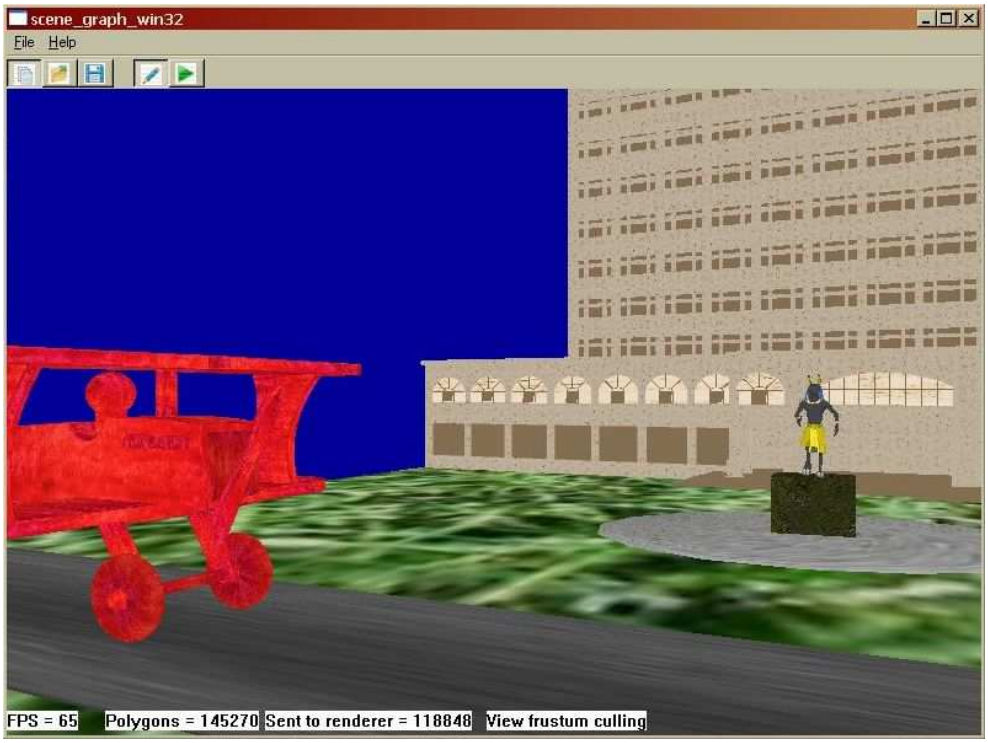
Na slici 12e sam malo zarotirao pogled u desno i kao rezultat dobio izuzetno povećanje performansi na oko 427 FPS-a. Tu kompletno odbacujem objekt hotela već u čvoru grafa scene. Mora se reći da se dio ubrzanja pripisuje i manjem broju ispitivanja čvorova, no većina ubrzanja nastaje zbog puno manjeg broja poligona poslanih na iscrtavanje.

Idući slučaj pokazuje pogled u scenu gdje dolazi do suprotnog efekta.

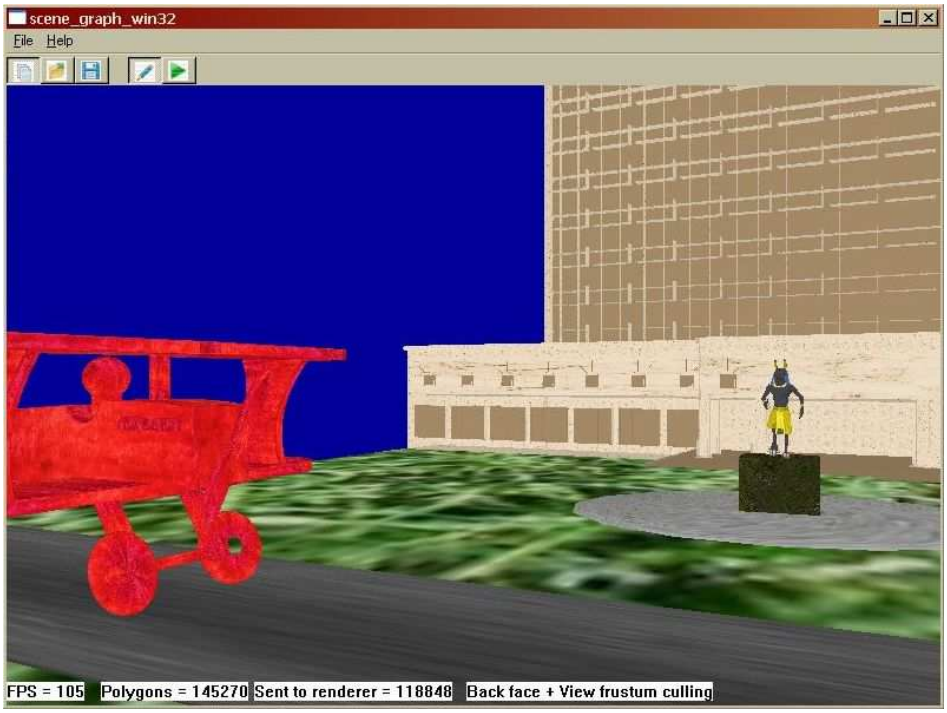
Slika 13a. Isključeni svi algoritmi.



Slika 13b. Odbacivanje stražnjih poligona.



Slika 13c. Odbacivanje po projekcijskom volumenu.



Slika 13d. Odbacivanje stražnjih poligona i odbacivanje po projekcijskom volumenu.

Algoritam	Broj poligona poslanih na iscrtavanje	FPS
Bez algoritma (13a)	145270	74
Odbacivanje stražnjih poligona (13b)	145270	114
Odbacivanje po projekcijskom volumenu (13c)	118848	65
Odbacivanje stražnjih poligona + odbacivanje po projekcijskom volumenu (13d)	118848	105

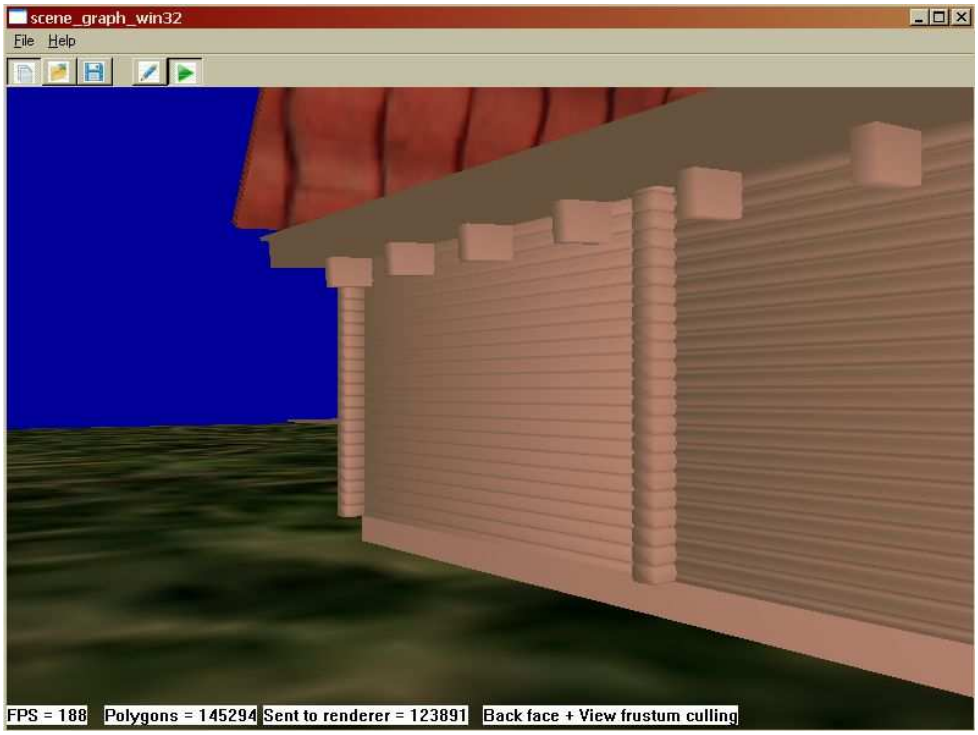
Tablica 2. Tablica uz sliku 13.

Ovo je najgori pogled u scenu što se tiče odbacivanja po projekcijskom volumenu. Projekcijski volumen presijeca dva najkompleksnija objekta u sceni. To su hotel i avion. Kip je isto kompleksan objekt no ne toliko i cijeli je u projekcijskom volumenu.

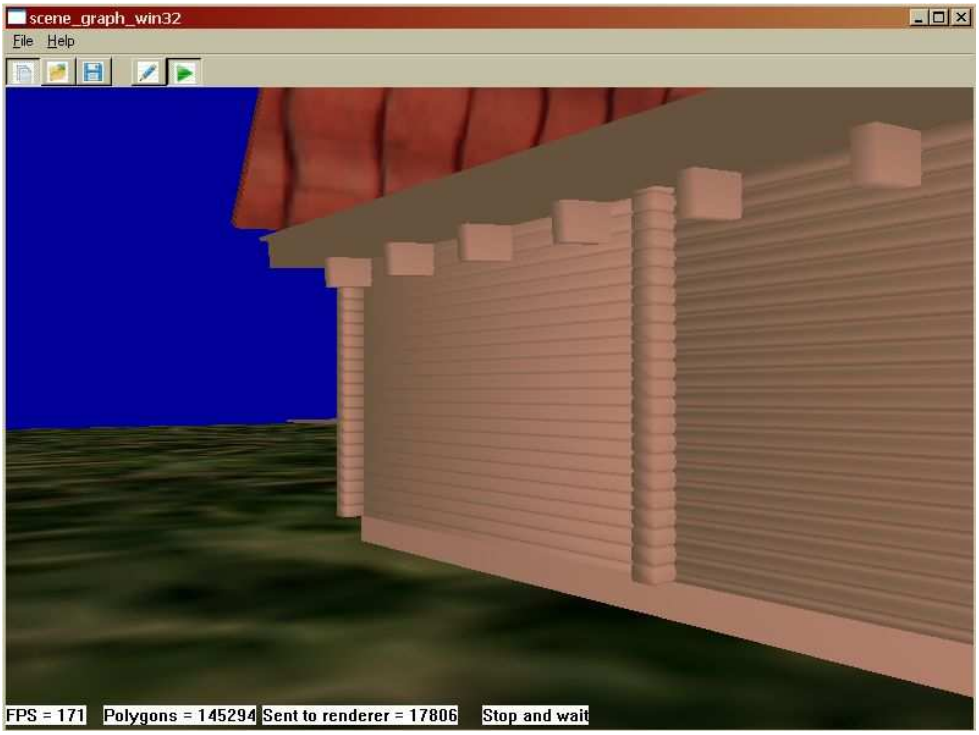
Po slikama 13a – 13d vidimo da je odbacivanje stražnjih poligona najzahvalniji način odbacivanja suvišnih poligona i uvijek će dovesti do određenog ubrzanja. U slučaju odbacivanja po projekcijskom volumenu to ne možemo reći. Pošto siječemo dva kompleksna objekta, testiranje prisutnosti u sceni postaje usporavajući faktor. Puno čvorova oba oktalna stabla se moraju ispitati i, ako volumen siječe čvor stabla, onda moramo ispitati njegovu djecu. To je razlog usporavanja iscrtavanja koji je prisutan na slikama 13c i 13d. Inače bi mogli dio usporavanja tražiti i u više glDrawElements pozivima, no ovdje je to zanemarivo. U ovakvim slučajevima može se dodati uvjet ispitivanja. Ako je broj čvorova koje trebamo ispitati toliko velik da je vrijeme njihovih ispitivanja veće od vremena koje dobijemo kod ubrzanja zbog odbacivanja, onda pošaljemo čitavu scenu na iscrtavanje. Vidimo da je, u ovom pogledu, vidljivo oko 80% scene, što je jako puno. Može se dodati uvjet - ako se vidi oko 75% scene, onda iscrtaj cijelu scenu.

6.2 “Stani i čekaj“ i CHC++ algoritam

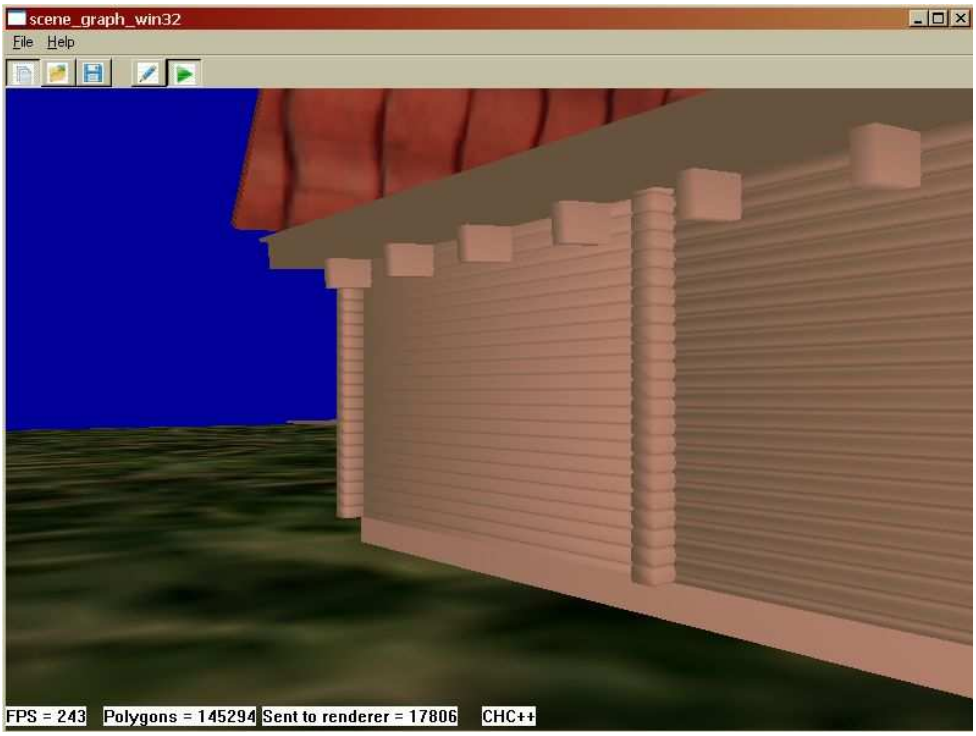
Za ova dva algoritma sam opet koristio istu scenu. Općenito svi algoritmi odbacivanja zaklonjenih poligona su najisplativiji u scenama zatvorenog prostora i u pogledima gdje je većina scene zaklonjena. “Stani i čekaj“ algoritam je koristan kada je skoro cijela scena zaklonjena, no većinom nudi slabije performanse čak i od samog odbacivanja po projekcijskom volumenu. CHC++ nudi veliki porast FPS-a u zaklonjenim pogledima i prihvatljiv pad FPS-a u pogledima gdje ništa nije zaklonjeno.



Slika 14a. Odbacivanje po projekcijskom volumenu.



Slika 14b. “Stani i čekaj“ algoritam.



Slika 14c. CHC++ algoritam.

Algoritam	Broj poligona poslanih na iscrtavanje	FPS
Odbacivanje po projekcijskom volumenu (14a)	123891	188
“Stani i čekaj“ (14b)	17806	171
CHC++ (14c)	17806	243

Tablica 3. Tablica uz sliku 14.

Na slikama 14a – 14c stoji primjer pogleda u scenu gdje kuća čini veliku okluziju scene. Vidimo da je broj poslanih poligona kod odbacivanja po projekcijskom volumenu preko 120000, dok je kod druga dva algoritma taj broj nešto veći od 17000. Znači da dolazi do značajnog odbacivanja suvišnih poligona. Broj od 188 FPS-a kod odbacivanja po projekcijskom volumenu je solidan uzevši u obzir broj poligona, ali kod "stani i čekaj" dobivamo pad za 15-ak FPS-a unatoč puno manjem broju poslanih poligona. Dakle vrijeme koje čekamo na odgovor upita o zaklanjanju je ipak preveliko da bi dobili ikakvo ubrzanje. CHC++ algoritam nam u ovom pogledu otrkiva svu svoju raskoš. Donosi ubrzanje u odnosu na odbacivanje po projekcijskom volumenu od gotovo 60 FPS-a. Ovo pripisujem velikom broju odbačenih poligona, ali i puno naprednijem algoritmu u odnosu na naivnu metodu. CHC++ ne troši vrijeme čekajući na odgovor na upit o zaklanjanju već dalje prolazi kroz graf i radi posao koji "stani i čekaj" algoritam radi tek kad dobije odgovor o upitu. Također vidljivi čvorovi se ne ispituju svaki trenutak prikaza nego svaki deseti, a nevidljivi čvorovi se grupiraju ovisno o odnosu broja čvorova i troška upita.

Na slikama 15a – 15c sam proučio primjer gdje nema zaklanjanja.

Slika 15a. Odbacivanje po projekcijskom volumenu.

Slika 15b. “Stani i čekaj“ algoritam.

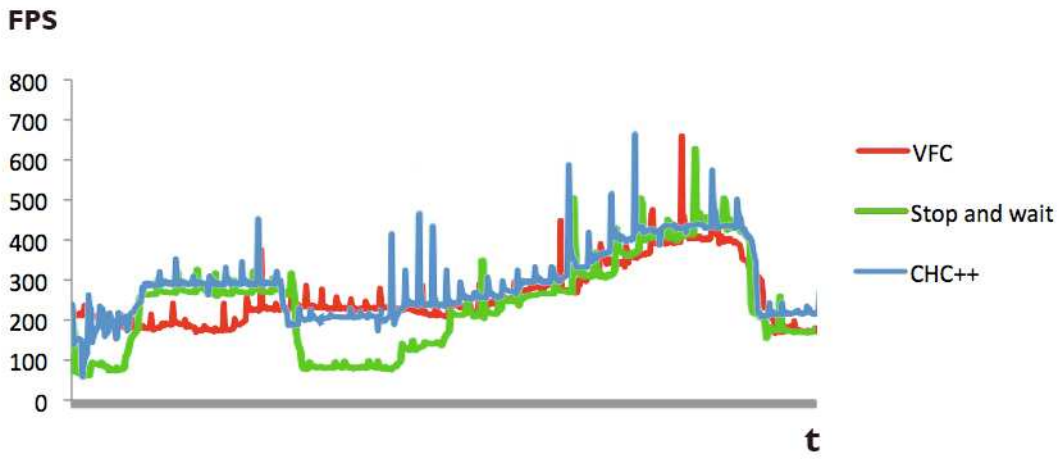
Slika 16c. CHC++ algoritam.

Algoritam	Broj poligona poslanih na iscrtavanje	FPS
Odbacivanje po projekcijskom volumenu (15a)	71942	220
“Stani i čekaj“ (15b)	71942	102
CHC++ (15c)	71942	212

Tablica 4. Tablica uz sliku 15.

U slučaju na slici 15 vidimo da je broj odbačenih poligona jednak za sva tri algoritma, a to znači da nema zaklonjenih objekata. Vidimo da najbolje performanse nudi algoritam za odbacivanje po projekcijskom volumenu, no tek nešto lošiji je CHC++. U CHC++ algoritmu svi vidljivi čvorovi se ispituju svaki deseti trenutak prikaza s time da se ne provjeravaju svi u istom trenutku prikaza. Za to je odgovoran slučajni odabir broja trenutka prikaza u kojem će se desiti prvi upit. Tako da se upiti raspodijele po različitim trenutcima prikaza i time ublaže trošak algoritma. Apsolutno najgore rezultate nudi naivna “stani i čekaj“ metoda. U svakom trenutku prikaza se iscrtavaju obujmice od svih vidljivh čvorova (ali ne na ekran), postavljaju se upiti, čekaju se odgovori na te upite i iscrtava se geometrija svih vidljivih čvorova.

Kod ovakvih algoritama je dosta bitan prolazak kroz scenu, a ne samo jedan pogled. Ja sam u tu svrhu odredio istu putanju prolaska za sva tri algoritma i usporedio dobivene rezultate. Dobiveni rezultati nalaze se na grafu 1.



Graf 1. Primjer prolaska kroz scenu s izmjerenim performansama.

Na početku grafa 1 vidimo dio gdje se CHC++ i “stani i čekaj“ algoritam odvajaju i nude puno bolje rezultate nego odbacivanje po projekcijskom volumenu. To je dio kad sam s kamerom ušao u kućicu u sceni i gotovo cijela scena je bila zaklonjena. U tom slučaju je i algoritam “stani i čekaj“ dosta dobar, malo lošiji od CHC++. To je zato što postoji samo jedan vidljiv čvor, a svi ostali se trebaju ispitati i treba pričekati njihove rezultate. Nakon toga sam s kamerom izašao iz kućice što se i očituje na grafu. CHC++ tu bilježi nagli pad FPS-a, no puno manji od naivne metode. To je otprilike onaj pogled koji sam objasnio na slici 16. Vidimo da je CHC++ lošiji od običnog odbacivanja po projekcijskom volumenu i taj pad iznosi oko 20-30 FPS-a. Do kraja prolaska su performanse svih algoritama otprilike iste jer sam krenuo prema najvećem objektu u sceni i samo je on bio vidljiv. Na samom kraju dolazim do stražnje strane modela hotela i okrećem kameru prema sceni. Vidi se da je tu CHC++ nešto bolji od ostala dva algoritma.

Na kraju bih htio napomenuti da, kod algoritama sa sklopovskim upitima, ne koristim oktalno stablo već samo obujmice grafa scene kao čvorove za slanje upita. Napravio sam i verziju sa oktalnim stablom, ali dobio sam malo lošije rezultate. Nevolja kod sklopovskih upita o zaklanjanju jest ta da, kad je objekt cijeli vidljiv, mi to ne znamo, već samo znamo broj vidljivih slikovnih elemenata. Onda moramo taj čvor rastaviti na njegovu djecu i isto napraviti za njih. Ako scena ima puno kompleksnih objekata, radi se velik posao koji oduzima nešto vremena i ispada da je bolje iscrtati čitav objekt.

Za vrijeme ovog rada sam sudjelovao u diskusijama i proučavao slične rasprave na internetu o tome koliko su korisni sklopovski upiti o zaklanjanju. Mišljenja su podijeljena. Govori se da je bolje koristiti što veće obujmice, što sam i ja zaključio. Spominje se samo par novih aplikacija u kojima se koriste i uglavnom se smatraju kao znanstvena tema otvorena za daljnji razvoj i poboljšanja. CHC i CHC++ algoritmi su došli korak bliže komercijalnoj upotrebi i smatraju se vodećim riješenjima kada je riječ o skopovskim upitima o zaklanjanju. Ovi algoritmi su vrlo korisni u scenama zatvorenog prostora, no u otvorenim scenama gdje ima malo ili nimalo zaklonjenih objekata su nepotrebni.

7. Zaključak

Kod grafa scene i optimizacija iscrtavanja nekoliko stvari je vrlo bitno. Graf scene mora biti prostorno i logički dobro organiziran. Ako imamo loše strukturiranu scenu, imamo loše temelje za daljnje optimizacije iscrtavanja. Na primjer možemo grupirati dva objekta na suprotnim stranama scene, a to rezultira obujmicom koja se prostire duž cijele scene i gotovo svaki pogled će presijecati tu obujmicu. Imati ćemo nepotrebna ispitivanja prisutnosti u sceni. Nadalje treba dobro analizirati algoritme, naći usko grlo i, na taj način, još više optimizirati iscrtavanje. To smo vidjeli u primjeru na slici 13 gdje je sam algoritam optimizacije usporavajući faktor.

Općenito kod programiranja aplikacija kod kojih nam je bitan FPS, moramo dobro analizirati svaki korak jer se gotovo svi algoritmi izvode svaki trenutak prikaza. Tu bih se osvrnuo na problem na koji sam naišao radeći na praktičnom dijelu ovog rada. Naime nakon što sam implementirao odbacivanje po projekcijskom volumenu, krenuo sam u analizu optimizacija. Iznenadio sam se kada sam vidio da ne dolazi do ubrzanja iscrtavanja. U nekim slučajevima sam primjetio i usporavanje. Tada sam uključio odbacivanje stražnjih poligona i dobio isti FPS kao i kada je bilo isključeno. Temeljitom analizom svake linije koda, a pogotovo OpenGL poziva, uz poseban program za analizu takvih aplikacija, zaključio sam gdje je usko grlo. To je bilo slanje vrhova na iscrtavanje. Velike količine podataka sam slao grafičkoj kartici svaki trenutak prikaza i to je dovelo do zbunjujućih i vrlo loših rezultata. Ovaj problem sam riješio tako da, na početku programa, šaljem čitavu scenu u memoriju na grafičkoj kartici, a jedino što mijenjam svaki trenutak prikaza su indeksi vrhova koji se trebaju iscrtati. OpenGL od verzije 1.5 nudi funkcije za pristup i baratanje sa spremnikom za vrhove (engl. Vertex buffer object) [19]. Nakon što sam se riješio tog uskog grla, napokon sam dobio razumljive rezultate pogodne za analizu.

Na kraju mogu zaključiti da je izrada programa za prikaz virtualne scene vrlo pipljiv, ali i zanimljiv posao kod kojeg treba pratiti što se događa u svakom trenutku prikaza, paziti da bude što manje ili nimalo redudantnih OpenGL poziva i uvijek težiti ka novim, bržim riješenjima što ih OpenGL nudi svakom nadogradnjom.

8. Literatura

[1] I. S. Pandžić, “Virtualna Okruženja“, 2004.

[2] P. Martz, “OpenSceneGraph Quick Start Guide“

[3] A. M. L. Karim, M. S. Karim, E. Ahmed, Dr. M. Rokonuzzaman, “Scene Graph Management for OpenGL based 3D Graphics Engine“, International Conference on Computer & Information Technology (ICCIT) 2003

[4] http://en.wikipedia.org/wiki/Kd-tree

[5] http://www.lighthouse3d.com/opengl/viewfrustum

[6] S. Teller, “Visibility Computations in Densely Occluded Environments”, University of California, Berkeley, 1992.

[7] S. Teller, C. Sequin, “Visibility preprocessing for interactive walkthroughs”, Computer Graphics, 1991.

[8] G. Schaufler, J. Dorsey, X. Decoret, F. Sillion, “Conservative Volumetric Visibility with Occluder Fusion”, SIGGRAPH 2000.

[9] D. Luebke, C. Georges, “Portals and mirrors: Simple, fast evaluation of potentially visible sets”, SIGGRAPH 1995.

[10] T. Hudson, D. Manocha, J. Cohen, M. Lin, K. Hoff, H. Zhang, “Accelerated occlusion culling using shadow frustra”, 1997

[11] J. Bittner, V. Havran, P. Slavik, “Hierarchical visibility culling with occlusion trees”, 1998.

[12] N. Greene, M. Kass, Gary Miller, “Hierarchical Z-buffer visibility”, 1993.

[13] J. Bittner, M. Wimmer, H. Piringer, W. Purgathofer, “Coherent hierarchical culling: Hardware occlusion queries made useful”, EUROGRAPHICS 2004.

[14] M. Guthe, A. Balazs, R. Klein: “Near optimal hierarchical culling: Performance driven use of hardware occlusion queries”, Eurographics Symposium on Rendering 2006.

[15] O. Mattausch, J. Bittner, M. Wimmer, “CHC++: Coherent Hierarchical Culling Revisited”, EUROGRAPHICS 2008.

[16] D. H. Eberly, “3D Game Engine Architecture”, 2005.

[17] D. Cohen-Or, Y. Chrysanthou, C. Silva, F. Durand, “A survey of visibility for walkthrough applications”, 2002.

[18] http://www.gamedev.net/

[19] http://www.songho.ca/opengl/gl_vbo.html#draw

[20] D. Shreiner, M. Woo, J. Neider, T. Davis, “OpenGL Vodič za programere“, 2007.